Dependent Communication in Type Theory

Nicolas Guenot, Daniel Gustafsson, and Nicolas Pouillard

IT University of Copenhagen, Denmark
{ngue,dagu,npou}@itu.dk

Abstract

We present an expressive type system for a language extending the usual λ-calculus with concurrency primitives from the π-calculus. This language uses dependent types as found in type theory, enriched with linear types for processes. It allows to define programs performing communication according to complex protocols, including possible dependencies on the communicated data. The core of this type system is Martin-Löf type theory, and it is extended with rules derived from linear logic. Finally, we present its formalisation as an embedding into AGDA and discuss the encoding of complex protocols.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs: Mechanical verification, F.4.1 Mathematical Logic: Lambda calculus and related systems

Keywords and phrases Type Theory, Linear Logic, Process Calculi, Dependent Types, Session Types

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Functional programming is a successful paradigm, that builds on the notion of types to allow for safe and expressive languages. The development of type theory [14] has extended the scope of types to the specification and verification of properties of programs, up to the point where one can write programs together with specifications that are all validated through a type system, not only from the perspective of proof assistants [6], and also from the viewpoint of pure programming, when using dependent types [16]. Beyond the original scope of functional programming, the representation of communication in programs has been intensively investigated over the last few decades, in particular with the development of process algebras [11] and the π-calculus [15]. This has lead to progress in the area of types for processes, in particular with the introduction of session types [12] and later with their connection to linear logic [4, 19]. However, the theory of typed communicating programs is not yet as developed as the theory of functional programs, and this leads naturally to the question of how one might integrate one into the other.

Among the possible combinations of the functional and communication-centric paradigms, we are interested here in the idea of extending a typed λ-calculus, seen as a model of functional programming, with primitives for communication, as done for example in the study of session types for functional programs [7]. However, the system we consider integrates communication of functional data as well as channel names, building on the connection between the π-calculus and linear logic and going further than previous integrations based on a similar scheme [17]. In particular, we use a full dependent type theory as host functional language. The difficulty lies in the smooth integration of the delicate linear typing of processes in a system where dependencies can appear between non-linear terms and types. The reward for such an integration is high, as it provides the means of defining communication protocols depending on the data transmitted, and ensures that these are implemented by well-behaved programs.

The association of communicating processes and dependent types has two complementary sides: it extends the expressive power of session types as used for variants of the π-calculus, and it simplifies the task of reasoning about communicating programs. This is well illustrated by our

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

choice of Martin-Löf type theory as host language, as this is the theory underlying the AGDA [16] language, intended as a framework for programming and reasoning using dependent types. The purpose of the language we propose here is to introduce a minimal extension to type theory: it shall support enough communication to avoid extending further the language, but rather encode the types necessary to express more complex interactions.

Linear types for processes. The cornerstone of this investigation is the introduction of the multiplicative fragment of linear logic [8], as presented in the sequent calculus, into Martin-Löf's type theory. In order to maintain the properties of linear typing of processes, we isolate processes from functional terms but define an interface to allow these two forms of programs to mix. Such a separation is essential, and is enforced by using two different typing judgements.

A notable difference between our system and related ones [17] based on the interpretation of linear formulas as sessions [4, 19] is that the linear connectives $⅋$ and $\otimes$ are used to organise channels rather than indicate input/output behaviour. The strictly behavioural part of the types we use consists in the standard input and output types from session types [18]. As a result, the processes typed in this system are those obtained from proofs through the translation described by Abramsky [1], extended with special inputs and outputs for functional terms. The particular use of linear rules to type processes and their integration in type theory are described in Section 2 and Section 3 respectively — note that basic knowledge of linear logic [8] is assumed.

Encoding processes in AGDA. On a more practical level, one of the interests of the extension to Martin-Löf's type theory proposed here is that it can be encoded in AGDA by introducing a new universe for *protocols* — collections of session types associated to channel names — and defining processes in the language of type theory. We describe in Section 4 such an encoding, and discuss the subtleties involved in the handling of linear proofs. This development also contains a proof of cut elimination in AGDA, ensuring that well-typed terms containing processes reduce correctly and without deadlock nor livelock. Furthermore, it forms a basis for future investigations into the use of processes inside type theory, or the mechanised verification of protocol specifications expressed in the language of type theory.

Representing complex protocols. Only a small fragment of linear logic is introduced into type theory here, where rules are purely multiplicative and thus contain no duplication, even in a restricted or controlled way. An interesting observation is that this, combined with the interface between intuitionistic type theory and linear logic, is enough to recover the specification of more complex behaviours, using for example controlled forms of duplication.

We study in Section 5 the encoding of communication protocols exploiting some possibilities of dependent types. For instance, the case analysis behaviour obtained by considering additives in concurrent interpretations of linear logic [4] can be recovered by making a session type depend on the value of a bit transmitted using the data communication primitives. We also discuss the use of our framework to specify distributed variants of algorithms which can be easily specified with strong guarantees in dependent type theory, such as sorting algorithms.

Limits to linearity in type theory. One should note that although the type theory we present here allows one to exploit the expressivity of dependent types in a language supporting processes and sessions, defining a dependent form of session types is a different matter entirely. Indeed, dependencies are handled at the level of intuitionistic types and we cannot specify a type *linearly depending* on a term, as it would require defining a precise meaning for such a dependency. The investigations conducted towards this end all face the difficulty of this question, and so far only limited forms of linearity have been mixed with dependent types, in the sense that the dependent product is disconnected from the linear decomposition of implication [5, 13]. Even though the modalities of such an integration are better understood now, the limits to the use of dependencies in a linear type theory have not yet been overcome.

$$\frac{\vDash P :: \mathcal{J},[d:S],[e:T]}{\vDash c(de)P :: \mathcal{J},[c:S \parr T]} \qquad \frac{\vDash P :: \mathcal{J},[d:S,e:T,\mathcal{G}]}{\vDash \overline{c}\langle de\rangle P :: \mathcal{J},[c:S\otimes T,\mathcal{G}]} \qquad \frac{\vDash P :: \mathcal{J},[\mathcal{H},\mathcal{G}]}{\vDash P :: \mathcal{J},[\mathcal{H}],[\mathcal{G}]} \qquad \frac{}{\vDash \mathbf{0} :: [\cdot]}$$

$$\frac{\vDash P :: \mathcal{I},[\mathcal{G}] \quad \vDash Q :: \mathcal{J},[\mathcal{H}]}{\vDash P \parallel Q :: \mathcal{I},\mathcal{J},[\mathcal{G},\mathcal{H}]} \qquad \frac{\vDash P :: \mathcal{J},[c:S,d:S^{\perp}]}{\vDash (\nu cd)P :: \mathcal{J}} \qquad \frac{}{\vDash c \longleftrightarrow d :: [c:S],[d:S^{\perp}]}$$

Figure 1 Typing rules for processes based on linear logic

2 Typing Processes with Linear Logic

The rules we will use to type processes are variations on the standard use of linear logic as a type system for the π-calculus, described in [1] and [3]. This interpretation of linear proofs has been later modified in both an intuitionistic linear [4] and a classical linear [19] variant, to follow the principles of *session types* [12], but we return here to the original design of Abramsky.

An important problem in such an interpretation of sequent calculus proofs lies in the handling of parallel composition and the corresponding branching in a proof. Indeed, if a communication operation is associated to a logical rule such as the right rule for $\otimes$, we will need to type both the operator and a parallel composition, as done for example following the translation of Abramsky:

$$\frac{\vDash P :: \Gamma, d:S \qquad \vDash Q :: \Delta, e:T}{\vDash \overline{c}\langle de\rangle\,(P \parallel Q) :: \Gamma,\Delta, c:S\otimes T}$$

where we have simplified the picture by using a binding output. This leads to a host of problems with the syntactic congruence on processes and the preservation of typing. In order to avoid this, we depart from the usual presentation of linear logic and enrich its sequent calculus with two levels, using typing judgements of the shape $\vDash P :: \mathcal{J}$, where:

$$\mathcal{I},\mathcal{J} ::= \cdot \mid [\mathcal{H}] \mid \mathcal{I},\mathcal{J} \qquad\qquad \mathcal{G},\mathcal{H} ::= \cdot \mid c:S \mid \mathcal{G},\mathcal{H}$$

and where c denotes a *channel* name, while S denotes a linear formula seen as the type of this channel. The intuitive interpretation of this two-level approach is simple: comma in $\mathcal{J}$ represents a meta-level $\parr$ while in $\mathcal{H}$ it represents a meta-level $\otimes$. The view of the comma as the $\parr$ is the standard interpretation of sequents in linear logic, but here we can indicate that formulas should be distributed into different sequents, using brackets $[\cdot]$. The purpose of this generalisation is to make branching — and the splitting of the context — independent from the decomposition of the connectives. From a proof-theoretical viewpoint, this is related to the need to decompose the formulas not only at top level, a question adressed partially using *hypersequents* [2] and more thoroughly studied in the setting of *deep inference* [10]. The form of nesting required here is quite limited, as it only offers access to formulas appearing under a $\parr$ and a $\otimes$, when considering the logical interpretation of sequents.

This approach leads to the system described in Figure 1, where processes can contain binary inputs and outputs, parallel compositions and scopes, as well as the inactive process and a form of forwarder found in [4]. An important observation here is that in the rule for parallel composition, both $\mathcal{G}$ and $\mathcal{H}$ can be $\cdot$ and thus the *mix* rule is present. Although it is not a problematic rule, it is not part of standard linear logic. Moreover, in order to obtain the expected properties for $\parallel$ we need the rule treating the composition of two blocks $[\cdot]$, which also amounts to the presence of mix, by interpretation of the judgements: the implication $(A\otimes B) \multimap (A \parr B)$ is valid here.

Polyadic communication. We can easily extend the rules for $⅋$ and $\otimes$ to support the n-ary variants of the multiplicative connectives, as follows:

$$\frac{\vDash P :: \mathcal{J}, \overrightarrow{[e : S]}}{\vDash c(\vec{e})\,P :: \mathcal{J}, [c : ⅋\vec{S}]} \qquad \frac{\vDash P :: \mathcal{J}, [\overrightarrow{e : S}, \mathcal{H}]}{\vDash \overline{c}\langle\vec{e}\rangle\,P :: \mathcal{J}, [c : \otimes\vec{S}, \mathcal{H}]}$$

and thereby obtain typing rules for general polyadic communication, as found in the π-calculus. Here, a concise syntax indicates the use of a whole sequence of blocks $[\cdot]$ or types of the shape $c : S$, extracted from the n-ary connective. Note that by doing so, we obtain the multiplicative units of linear logic for free, but we are not interested in the distinction between the two. Beyond accepting the implication induced by mix, we simply collapse the two units, so that the rule for $\mathbf{0}$ can be written:

$$\frac{}{\vDash \mathbf{0} :: \cdot}$$

and the distinction beetween the *nullary* case of $⅋$ and $\otimes$ will not be emphasised in the following, since they are now logically equivalent. Observe that extending linear logic with the mix rule is neither problematic nor surprising, since the concurrent interpretations of linear logic are related to the use of proof-nets [9], as observed in [3], where mix simply translates as the juxtaposition of valid nets. The collapse of the two multiplicative units makes sense when considering them as types simply indicating termination of a process.

There is one rule in the system described in Figure 1 that is not reflected in the structure of processes, which is not fully satisfactory from a *proofs-as-programs* perspective. However, this could be avoided by using a more complicated variant of the rule for $\parallel$ where several blocks $[\cdot]$ blocks are split, under some conditions on groupings chosen in the premises. We keep the rules separated for the sake of clarity in this presentation.

Finally, note that we use a binary form of scoping, reminiscent of [18] but introducing two distinct channel names c and d. This stems from the design of the *cut* rule chosen here, where no branching is performed: this perspective on cut is consistent with the treatment of $\otimes$ in a system where depth is used and cut is essentially viewed as the implication $(A \otimes A^{\perp}) \multimap \perp$. Indeed, we wish to avoid repetition of channel names inside a sequent, while grouping dual session types of the cut into a $[\cdot]$ block.

Congruence and permutations. The decoupling of $\otimes$ and of cut from branching allows us to clarify the interpretation of each rule as a single construct in the π-calculus, but also provides a natural interpretation of the syntactic congruence $\equiv$ on processes in terms of permutations of inference rules. Indeed, in the sequent calculus and in the generalisation we consider here, many rules can be exchanged without changing the essential structure of the proof. Moreover, such permutations are necessary in the cut elimination process, which proceeds by permuting cut instances upwards in a proof tree.

The acknowledgement of rule permutations has two consequences. First, we can derive the expected equations on $\parallel$ from permutations of its rule with itself and others:

$$P \parallel Q \;\equiv\; Q \parallel P \qquad P \parallel (Q \parallel R) \;\equiv\; (P \parallel Q) \parallel R \qquad P \parallel \mathbf{0} \;\equiv\; P \qquad (\nu cd)(P \parallel Q) \;\equiv\; (\nu cd)P \parallel Q$$

under the condition that neither c nor d appear in Q in the last equation. Then, permutations involving the rules for $⅋$ and $\otimes$ justify the syntax of prefixes where no indication of sequentiality appears, just as in [3], since exchange of prefixes is possible. This means that we can validate the equations $\pi\kappa P \equiv \kappa\pi P$ if π and κ are *involving fully distinct names*, and $(\nu cd)\,\pi P \equiv \pi(\nu cd)\,P$ if c and d do not appear inside π. Moving prefixes past each other is unusual in the π-calculus, but it is necessary from a logical viewpoint to ensure that no cut can reach a blocked situation preventing its elimination.

Cut elimination and communication. In a type system for π based on the sequent calculus, cut elimination appears at the level of processes as reduction, specified through a rewriting system performing communication steps in appropriate configurations. The multiplicative fragment has a simple cut elimination procedure, with a single principal case involving $\parr$ and its dual $\otimes$. The cut reduction in that case translates on processes as the rewrite rule:

$$(\nu cd)(\bar{c}\langle c_0 c_1\rangle P \parallel d(d_0 d_1) Q) \quad \to \quad (\nu c_0 d_0)(\nu c_1 d_1)(P \parallel Q)$$

which corresponds to the usual communication rule of the π-calculus, when considering binary scoping as an alternative to channel renaming. Beyond the simple reorganisation of the channel connections appearing in a process, the integration of this system into type theory will provide the means of communicating functional data, thus moving closer to the setting of a functional programming language extended with communication [7].

3 Communication in Dependent Type Theory

We now turn to the complete system, incorporating processes into dependent type theory. This system is based on the version of Martin-Löf type theory underlying Agda [16]. Our methodology is to extend the syntactic categories of terms involved with both a term level for communicating programs, using a syntax coming from the π-calculus [15], and a type level of *sessions* inspired by both linear logic [8] and session types [18]. The resulting type theory, including processes, will be called ***PTT*** in the following.

► **Definition 1.** The *terms, processes, types* and *sessions* of the ***PTT*** type theory are defined by t, P, A and S respectively in the following grammar:

$$\begin{array}{rcl} t,u & ::= & x \mid a \mid \lambda(x\!:\!A).t \mid t\;u \mid \langle t,u\rangle \mid \bullet \mid \pi_1\,t \mid \pi_2\,t \mid \vec{c}.P \mid A \mid S \\ P,Q & ::= & \mathbf{0} \mid P \parallel Q \mid (\nu cd)P \mid c(\vec{e})P \mid \bar{c}\langle\vec{e}\rangle P \mid c(x\!:\!A)P \mid \bar{c}\langle t\rangle P \mid t \mathbin{@} \vec{c} \\ A,B & ::= & t \mid \Pi(x\!:\!A).B \mid \Sigma(x\!:\!A).B \mid \mathbb{1} \mid \{\!\!|\vec{S}|\!\!\} \mid \mathbf{Set}_i \mid \mathbf{Session}_i \\ S,T & ::= & t \mid ?(x\!:\!A).S \mid !(x\!:\!A).S \mid \parr\vec{S} \mid \otimes\vec{S} \end{array}$$

where letters such as x, y and z denote term *variables* and a denotes a *constant*, while c, d and e denote *channels* used in processes.

A large part of this syntax is standard for type theory, but functional terms are extended with the binding construct $\vec{c}.P$ representing the process P interacting on the channels recorded in the sequence $\vec{c}$. This can be viewed as the *packaging* of a process in a functional form, where no free channel name is allowed, as this would correspond to *dangling* connections from a programming perspective, and lead to problems during reduction.

The part of the syntax intuitively representing types is also extended, with a construct $\{\!\!|\vec{S}|\!\!\}$ in which a sequence of sessions is mentioned. Note that order matters in this operator, since these sessions will describe the channels recorded in the sequence $\vec{c}$ in the term $\vec{c}.P$. Moreover, we add the type $\mathbf{Session}_i$ representing the universe of sessions: this type, just as $\mathbf{Set}_i$, is annotated with a *level* in order to distinguish the levels of the type hierarchy, and we will consider an additive treatment of levels.

The fragment of the ***PTT*** system dealing with functional terms and types is essentially the standard type theory from Agda [16] with additional rules for the extra universes. The typing rules are shown in Figure 2, where $\Gamma \vdash t : A$ denotes a typing judgement associating type A to the term t in a context Γ, which is a list of typing hypotheses. The alternate judgement $\Gamma \vdash \cdot$ corresponds to the verification of well-formation for Γ, ensuring that each assumption mentions a type valid at that position in the context.

$$\frac{\Gamma \vdash \cdot \quad \Gamma \vdash A : \textbf{\textit{Set}}_i}{\Gamma, x : A \vdash \cdot} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vdash \textbf{\textit{Set}}_i : \textbf{\textit{Set}}_{i+1}} \qquad \frac{\Gamma \vdash A : \textbf{\textit{Set}}_i \quad \Gamma, x : A \vdash B : \textbf{\textit{Set}}_j}{\Gamma \vdash \Pi(x : A).B : \textbf{\textit{Set}}_{i \sqcup j}}$$

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vdash \mathbb{1} : \textbf{\textit{Set}}_0} \qquad \frac{\Gamma \vdash A : \textbf{\textit{Set}}_i \quad \Gamma, x : A \vdash B : \textbf{\textit{Set}}_j}{\Gamma \vdash \Sigma(x : A).B : \textbf{\textit{Set}}_{i \sqcup j}}$$

$$\frac{\Gamma \vdash \cdot \quad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash A \simeq B}{\Gamma \vdash t : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash \cdot \quad a : A \in \mathcal{S}}{\Gamma \vdash a : A} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \pi_1\, t : A} \qquad \frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{u/x\}}$$

$$\frac{\Gamma \vdash \cdot}{\Gamma \vdash \bullet : \mathbb{1}} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \pi_2\, t : B\{\pi_1\, t/x\}} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B\{t/x\}}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B}$$

Figure 2 Functional terms and types fragment of the ***PTT*** system

In Figure 2, we denote by $\mathcal{S}$ the *signature*, containing typed constants, that we leave implicit in all rules since it is specified externally. The rules for dependent products and sums involve the substitution of a term t for a variable x in a type A, denoted by $A\{t/x\}$. Moreover, the *conversion* rule allowing to consider normal forms of types is based on a relation defined externally.

▶ **Definition 2.** The *conversion* relation $\simeq$ between ***PTT*** terms t and u under a given Γ, written $\Gamma \vdash t \simeq u$, is defined as the smallest congruence such that:

$$\frac{t \to u}{\Gamma \vdash t \simeq u} \qquad \frac{\Gamma \vdash t : \Pi(x : A).B}{\Gamma \vdash t \simeq \lambda(x : A).t\ x} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash t \simeq \langle \pi_1\, t, \pi_2\, t \rangle} \qquad \frac{\Gamma \vdash t : \mathbb{1}}{\Gamma \vdash t \simeq \bullet}$$

where $\to$ denotes the reduction relation that we will define on ***PTT*** terms.

The verification of well-formedness of types, shown at the top of Figure 2, deals with levels of types using the notation $i \sqcup j$ for the maximum of i and j, and Figure 3 has a rule stating that the type $\textbf{\textit{Session}}_i$ has itself type $\textbf{\textit{Set}}_{i+1}$ so that this type can itself be used in the theory. In particular, one can write terms computing sessions, for which typing requires knowing the relation between the levels of ***Session*** and ***Set***.

Finally, the verification that $\{\!|\, \vec{S} \,|\!\}$ is a valid type relies on the rule checking that every single session S_k it contains is a valid session. In this rule, the notation $\sqcup \vec{\imath}$ denotes the maximum of a sequence of levels — the sequence obtained from the premises verifying each of the S_k sessions.

Processes and protocols. The typing rules for the fragment of ***PTT*** where processes, sessions and protocols — collections of sessions — appear are shown in Figure 3. Most rules are defining a judgement $\Gamma \vDash P :: \mathcal{J}$, where $\mathcal{J}$ is a two-level structure as defined in the previous section:

$$\mathcal{I}, \mathcal{J} ::= \cdot \mid [\mathcal{H}] \mid \mathcal{I}, \mathcal{J} \qquad\qquad \mathcal{G}, \mathcal{H} ::= \cdot \mid c : S \mid \mathcal{G}, \mathcal{H}$$

considered under the equations making $\cdot$ a unit for comma, and ensuring the commutativity and associativity of commas. Note that this structure is entirely at the meta-level in the type system, and is not part of the syntax of ***PTT***.

$$\frac{\Gamma \vdash \cdot}{\Gamma \vdash \textit{\textbf{Session}}_i : \textit{\textbf{Set}}_{i+1}} \qquad \frac{(\Gamma \vdash S_k : \textit{\textbf{Session}}_{i_k})_k}{\Gamma \vdash ⅋\vec{S} : \textit{\textbf{Session}}_{\sqcup\vec{i}}} \qquad \frac{\Gamma \vdash A : \textit{\textbf{Set}}_i \quad \Gamma \vdash S : \textit{\textbf{Session}}_j}{\Gamma \vdash ?(x:A).S : \textit{\textbf{Session}}_{i\sqcup j}}$$

$$\frac{(\Gamma \vdash S_k : \textit{\textbf{Session}}_{i_k})_k}{\Gamma \vdash \lBrace \vec{S} \rBrace : \textit{\textbf{Set}}_{\sqcup\vec{i}}} \qquad \frac{(\Gamma \vdash S_k : \textit{\textbf{Session}}_{i_k})_k}{\Gamma \vdash \otimes\vec{S} : \textit{\textbf{Session}}_{\sqcup\vec{i}}} \qquad \frac{\Gamma \vdash A : \textit{\textbf{Set}}_i \quad \Gamma \vdash S : \textit{\textbf{Session}}_j}{\Gamma \vdash !(x:A).S : \textit{\textbf{Session}}_{i\sqcup j}}$$

$$\frac{\Gamma \vDash P :: \mathcal{J}, [\overrightarrow{e:S}]}{\Gamma \vDash c(\vec{e})\,P :: \mathcal{J}, [c : ⅋\vec{S}]} \qquad \frac{\Gamma \vDash P :: \mathcal{J}, [\overrightarrow{e:S}, \mathcal{H}]}{\Gamma \vDash \bar{c}\langle\vec{e}\rangle\,P :: \mathcal{J}, [c : \otimes\vec{S}, \mathcal{H}]} \qquad \frac{\Gamma \vDash P :: \mathcal{J}, [\mathcal{H}, \mathcal{G}]}{\Gamma \vDash P :: \mathcal{J}, [\mathcal{H}], [\mathcal{G}]}$$

$$\frac{\Gamma \vDash P :: \mathcal{I}, [\mathcal{G}] \quad \Gamma \vDash Q :: \mathcal{J}, [\mathcal{H}]}{\Gamma \vDash P \parallel Q :: \mathcal{I}, \mathcal{J}, [\mathcal{G}, \mathcal{H}]} \qquad \frac{\Gamma \vDash P :: \mathcal{J}, [c : S, d : T] \quad \Gamma \vdash S \simeq_\perp T}{\Gamma \vDash (\nu cd)\,P :: \mathcal{J}} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vDash \mathbf{0} :: \cdot}$$

$$\frac{\Gamma \vdash A : \textit{\textbf{Set}}_i \quad \Gamma, x : A \vDash P :: \mathcal{J}, [c : S]}{\Gamma \vDash c(x : A)\,P :: \mathcal{J}, [c : ?(x:A).S]} \qquad \frac{\Gamma \vdash t : \lBrace \vec{S} \rBrace}{\Gamma \vDash t \mathbin{@} \vec{c} :: \overrightarrow{[c:S]}}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vDash P :: \mathcal{J}, [c : S\{t/x\}]}{\Gamma \vDash \bar{c}\langle t\rangle\,P :: \mathcal{J}, [c : !(x:A).S]} \qquad \frac{\Gamma \vDash P :: \overrightarrow{[c:S]}}{\Gamma \vdash \vec{c}.P : \lBrace \vec{S} \rBrace}$$

Figure 3 Process fragment of the ***PTT*** system

The rules for polyadic input and (bound) output from the π-calculus, parallel composition and scoping are directly imported from the system we considered in Section 2. However, the treatment of duality is more subtle here, and performed through a special relation.

▶ **Definition 3.** The *duality* relation $\simeq_\perp$ on sessions of ***PTT*** is the smallest symmetric relation containing the relation defined by the following rules:

$$\frac{\Gamma, x : A \vdash S \simeq_\perp T}{\Gamma \vdash ?(x:A).S \simeq_\perp !(x:A).T} \qquad \frac{(\Gamma \vdash S_k \simeq_\perp T_k)_k}{\Gamma \vdash ⅋\vec{S} \simeq_\perp \otimes\vec{T}} \qquad \frac{\Gamma \vdash S \simeq S' \quad \Gamma \vdash S' \simeq_\perp T}{\Gamma \vdash S \simeq_\perp T}$$

The core π-calculus is extended in ***PTT*** with operators for the input and output of functional data, and we write $c(x:A)$ and $\bar{c}\langle t\rangle$ for a reception on a channel c of a value of type A named x in the continuation, and an emission of t on c, respectively. The corresponding typing rules are best compared to those of the functional language from [19], inspired by [7], but incorporate the additional treatment of dependent types.

Beyond the interaction between functional terms and processes in data input and output, the interface between judgments $\vdash$ and $\vDash$ offers the possibility of combining these paradigms. From one to the other, *packaging* processes as terms and *unpackaging* terms to processes is trivial but for one question: that of the treatment of channel names in a protocol $\mathcal{J}$. The solution adopted here is to select an order among the free channel names when packaging, and to preserve that order in the functional layer until it is unpackaged following the same order. Notice that typing $t \mathbin{@} \vec{c}$ requires that blocks in the protocol contain a single session type, implying that compound blocks handled by cut and the $\otimes$ rule must be split within the phase where the corresponding rule appears. This allows the functional type $\lBrace \vec{S} \rBrace$ to remain a simple sequence of sessions.

Congruence and reduction. As mentioned in Section 2, the correspondence of reductions in a process calculus to the dynamics of cut elimination crucially relies on the identification of a number of syntactically distinct processes that are, for all intents and purposes, the same. The language of ***PTT*** is therefore equipped with an equivalence relation dealing with basic properties of $\|$ as well as the exchange of unrelated prefixes. This last notion is made precise by considering channel names in prefixes, and stating that input/output prefixes π and κ are *orthogonal*, which is denoted by $\pi \perp \kappa$, if and only if *they have no channel name in common*.

► **Definition 4.** The equivalence $\equiv$ on terms and processes of ***PTT*** is the smallest congruence relation satisfying the equations below:

$$\begin{array}{llll} P \,\|\, \mathbf{0} \equiv P & P \,\|\, (Q \,\|\, R) \equiv (P \,\|\, Q) \,\|\, R & (\nu cd)(P \,\|\, Q) \equiv (\nu cd)P \,\|\, Q & (c,d \notin Q) \\ P \,\|\, Q \equiv Q \,\|\, P & \pi\kappa P \equiv \kappa\pi P \quad (\pi \perp \kappa) & (\nu cd)\,\pi P \equiv \pi(\nu cd)P & (c,d \perp \pi) \end{array}$$

Although the equivalence considered here is close to the one used in an untyped π-calculus, it has no equation to remove unnecessary scope restrictions. Indeed, the logical correspondence of this construct to the cut rule prevents this, as a cut cannot be introduced or eliminated silently — because the definition of sessions prevents cuts on a symbol like $\cdot$ or an equivalent.

The next step is to equip the language of ***PTT*** with a reduction relation, defined as a rewriting system. The semantics given to the standard part of the type theory is as usual β-reduction, while the reductions involving communication operators correspond to the reduction of processes in the π-calculus. We have one additional reduction that treats the unpackaging of processes packaged as terms, but it is a simple unboxing operation.

► **Definition 5.** The reduction relation $\rightarrow$ on ***PTT*** terms and processes is the contextual closure of the following set of rewrite rules:

$$\begin{array}{lll@{\qquad}lll} (\lambda x.t)\,u & \rightarrow & t\{u/x\} & (\vec{c}.P) \,@\, \vec{e} & \rightarrow & P\{\vec{e}/\vec{c}\} \quad \text{if } |\vec{c}| = |\vec{e}| \\ \pi_1\,\langle t,u\rangle & \rightarrow & t & (\nu cd)(\overline{c}\langle\vec{e}\rangle\,P \,\|\, d(\vec{g})Q) & \rightarrow & (\nu\vec{e}\vec{g})(P \,\|\, Q) \\ \pi_2\,\langle t,u\rangle & \rightarrow & u & (\nu cd)(\overline{c}\langle t\rangle\,P \,\|\, d(x)Q) & \rightarrow & (\nu cd)(P \,\|\, Q\{t/x\}) \end{array}$$

We now have a complete system, and we turn to the properties it satisfies. First, we consider the stability of typing under the syntactic congruence defined to identify processes. This result is rather straightforward here, but it does not generally holds in session type systems based on linear logic [4, 19], where the grouping of outputs and parallel composition is problematic, for example, and restricts the shape of well-typed processes.

► **Theorem 6.** *For any terms t and u of* ***PTT****, if $\Gamma \vdash t : A$ and $t \equiv u$ then $\Gamma \vdash u : A$.*

Proof. By inspection of the equations for $\equiv$. In each case, we reorganise the typing derivation to obtain a new derivation for the equivalent term. Note that $\equiv$ is homomorphic on the functional fragment of the theory, and modifies only the parts of derivations corresponding to linear logic rule instances, which admit permutations with other independent rule instances. ◄

Then, we show that the system has subject reduction, so that typing is stable under reduction:

► **Theorem 7.** *For any terms t and u of* ***PTT****, if $\Gamma \vdash t : A$ and $t \rightarrow u$ then $\Gamma \vdash u : A$.*

Proof. By induction on the structure of the reduction from t to u, and by analysis of the shape of the typing derivation for t. In the case where the subterm reduced in t appears at top level, a case analysis shows that the reduction to u corresponds to a case from either normalisation in the type theory, or cut elimination in the linear system, based on the sequent calculus. ◄

More details on the normalisation result, in particular concerning the linear system where cut elimination corresponds to reduction, can be found in the next section, described from the viewpoint of the AGDA formalisation of ***PTT***.

4 Embedding Processes in the Agda language

The type theory presented in the previous section is based on the idea that a layer of processes can be introduced into a type theory *à la* Martin-Löf, thus allowing a calculus modelling concurrent programming to use dependent types. We now consider the realisation of this idea on a more practical level, in a dependently typed programming language: a part of the ***PTT*** language has been implemented[1] in AGDA [16], using a *shallow embedding* where the functional layer of ***PTT*** is directly translated as the corresponding part of the AGDA language.

Embedding the process layer in a functional language requires to define a representation that supports the properties expected from processes, but we have designed rules exploiting a rather general syntax to allow for more flexibility. In the AGDA representation, we simplify the theory by considering binary forms of $\parr$ and $\otimes$, and replace the cut and $\otimes$ rules by the branching variants:

$$\frac{\Gamma \vDash P :: \mathcal{I},[c:S] \quad \Gamma \vDash Q :: \mathcal{J},[d:T] \quad S \simeq_{\perp} T}{\Gamma \vDash (\nu cd)(P \parallel Q) :: \mathcal{I},\mathcal{J}} \qquad \frac{\Gamma \vDash P :: \mathcal{J},[d:S] \quad \Gamma \vDash Q :: \mathcal{J},[e:T]}{\Gamma \vDash \overline{c}\langle de\rangle\,(P \parallel Q) :: \mathcal{I},\mathcal{J},[c:S\otimes T]}$$

Moreover, we drop the *pure mix* rule and retain only, beyond the two rules above, the binary $\parr$ rule, the axiom typing $\mathbf{0}$, the mix rule typing $\parallel$ by branching and the rules for data input and output as well as the interfaces to the functional layer. The resulting system is equivalent to the original one on a logical level, except for the units. Indeed, n-ary connectives can be recovered in all cases where $n \geq 1$ and the pure mix rule was used because of the congruence only.

Data input and output. Interestingly, the rules introducing dependencies in a session type can be interpreted in a straightforward way inside pure type theory, building on the similarity of ? and ! to Π and Σ respectively. A product $\Pi(x:A).B$ can indeed interact with a sum $\Sigma(x:A).C$ in a way similar to input and output: the sum provides a proof of A and B assumes the existence of a proof x of A. These constructions are thus simple to handle in the AGDA representation.

Cut elimination in linear logic. The most important property of the ***PTT*** system is that any term can be reduced to a normal form. However, this requires to normalise terms containing processes and thus to showing that linear typing implies that communication can be properly executed in processes. This amounts to a proof of cut elimination in the multiplicative fragment of linear logic extended with mix, and accomodating the rules interfacing functional terms and processes. We discuss now the proof of cut admissibility used in the AGDA representation.

The proof proceeds by induction on the cut formula, and relies on *splitting lemmas*, in which a continuation term is inserted in a process to modify the subprocess involving the channel on which cut was performed. There is a splitting lemma for each possible shape of the session, but we show here only the case of $\otimes$:

$$\textit{(splitting lemma for } \otimes\textit{)} \quad \begin{array}{c} \qquad\qquad\qquad\qquad \vDash Q_0 :: \mathcal{I}_0,[d:S] \quad \vDash Q_1 :: \mathcal{I}_1,[e:T] \\ \qquad\qquad\qquad\qquad \vdots \\ \vDash P :: \mathcal{I},[c:S\otimes T] \qquad\qquad \vDash R :: \mathcal{I}_0,\mathcal{I}_1,\mathcal{J} \\ \hline\hline \vDash P\{R \mathbin{/\!\!/} \overline{c}\langle de\rangle\,(Q_0 \parallel Q_1)\} :: \mathcal{I},\mathcal{J} \end{array}$$

where the notation indicates that we modify some process P using a second process R *parametric* in the structures $\mathcal{I}_0$ and $\mathcal{I}_1$. The key idea is that inside P, there exists a subprocess of the shape $\overline{c}\langle de\rangle\,(Q_0 \parallel Q_1)$ corresponding to the decomposition of $[c:S\otimes T]$. The notation $\{\cdot \mathbin{/\!\!/} \cdot\}$, in the

[1] `https://github.com/crypto-agda/protocols`

style of a substitution, denotes the replacement of this output process by R where Q_0 and Q_1 are plugged. This splitting rule commutes upwards with all other rules until the adequate $\otimes$ rule is met — and this rule is unique by linearity.

The splitting lemma for $\parr$, the dual of $\otimes$, is expressed by the following scheme, where R is parametric in a single structure $\mathfrak{I}_0$ and contains only one open premise Q_0:

$$\textit{(splitting lemma for } \parr\textit{)}\quad \begin{array}{c} \begin{array}{cc} & \vDash Q_0 :: \mathfrak{I}_0, [d : S], [e : T] \\ & \vdots \\ \vDash P :: \mathfrak{I}, [c : S \parr T] & \vDash R :: \mathfrak{I}_0, \mathfrak{J} \end{array} \\ \hline\hline \vDash P\{R \,/\!\!/\, c(de)Q\} :: \mathfrak{I}, \mathfrak{J} \end{array}$$

and the combination of these two lemmas yields the case of cut reduction involving a formula $S \otimes T$ and its dual. Notice that no functional context Γ is mentioned here, since by the time normalisation requires eliminating a cut, this context is empty — as a consequence of reduction not being performed under abstractions in type theory. Consider a cut on $\otimes$ and $\parr$:

$$\dfrac{\vDash P :: \mathfrak{I}, [c : S \otimes T] \quad \vDash P' :: \mathfrak{J}, [d : S^\perp \parr T^\perp]}{\vDash (\nu cd)(P \parallel P') :: \mathfrak{I}, \mathfrak{J}}$$

that we rewrite into two cuts, on the two subformulas of these connectives, where the rightmost premise is an unknown process Q' typed using an unknown structure $\mathfrak{J}'$ that we introduce in the conclusion:

$$\dfrac{\vDash Q_0 :: \mathfrak{I}_0, [e : S] \qquad \dfrac{\vDash Q_1 :: \mathfrak{I}_1, [f : T] \quad \vDash Q' :: \mathfrak{J}', [g : S^\perp], [h : T^\perp]}{\vDash (\nu fh)(Q_1 \parallel Q') :: \mathfrak{I}_1, \mathfrak{J}', [g : S^\perp]}}{\vDash R \triangleq (\nu eg)(Q_0 \parallel (\nu fh)(Q_1 \parallel Q')) :: \mathfrak{I}_0, \mathfrak{I}_1, \mathfrak{J}'}$$

to obtain a process that we name R, parametric in the unknowns Q' and $\mathfrak{J}'$. We can now use this R to produce the expected proof, using a combination of the splitting lemmas for $\otimes$ and $\parr$:

$$\begin{array}{c} \vDash P :: \mathfrak{I}, [c : S \otimes T] \qquad \begin{array}{c} \vDash P' :: \mathfrak{J}, [d : S^\perp \parr T^\perp] \quad \vDash R :: \mathfrak{I}_0, \mathfrak{I}_1, \mathfrak{J}' \\ \hline\hline \vDash P'\{R \,/\!\!/\, d(gh)Q'\} :: \mathfrak{I}_0, \mathfrak{I}_1, \mathfrak{J} \end{array} \\ \hline\hline \vDash P\{P'\{R \,/\!\!/\, d(gh)Q'\} \,/\!\!/\, \bar{c}\langle ef\rangle (Q_0 \parallel Q_1)\} :: \mathfrak{I}, \mathfrak{J} \end{array}$$

This procedure can be applied the other rules, and we can therefore prove that all cuts can be eliminated from a typing derivation of ***PTT***. As a consequence, functional reductions blocked by some process constructs can be performed once the process has been reduced. Note that proving normalisation in ***PTT*** requires proving normalisation in the functional layer and cut elimination in the process layer simultaneously.

Forwarders. The process layer of the ***PTT*** system has no correspondent to the identity axiom, which was interpreted as a forwarder in Section 2, following [4]. However, we can express this process as detailed in Section 5.

5 Encoding Complex Protocols

In this section we illustrate the use of ***PTT*** through a series of examples and constructions. For this we use a type theory extended with inductively defined data types and data families, which is essentially the framework available in AGDA. Moreover, we consider in the following examples a framework extended with the features of ***PTT***, to illustrate their use in a practical setting. The examples are presented in the syntax of this slightly fictional extension of AGDA, and we use colors to distinguish the `names` of our definitions and the general `syntax` of the language. The following definitions are assumed:

- the empty type $\mathbb{0}$ represents falsity,
- the type $\mathbb{2}$ has two values 0_2 and 1_2 used to denote a single bit of information, but also as a boolean value, where 0_2 is false and 1_2 is true,
- these data types can be eliminated, and we write $\texttt{D-elim}\ \overrightarrow{a_i}\ d$ where D is the name of the type and each a_i is of some type A, and this eliminator maps each value d_i of the type D to a corresponding a_i.

Note that $\mathbb{2}$`-elim` corresponds to an `if` expression, and $\mathbb{0}$`-elim` is supposed to return a value of an arbitrary type, if provided with a value of the unhabited type $\mathbb{0}$. The type `_≡_` is the type of propositional equality, also called the identity type. AGDA reserves the usual equality symbol `=` for definitions, and we will apply this convention as well.

Additive connectives. Given two sessions S_0 and S_1 it is common to combine them as a *choice*. We use the standard notation from linear logic: a process p_0 following session $S_0 \oplus S_1$ chooses to follow either S_0 or S_1. Conversely a process p_1 following the session ${S_0}^{\perp} \mathbin{\&} {S_1}^{\perp}$ is responding to a choice between sessions ${S_0}^{\perp}$ and ${S_1}^{\perp}$, and thus needs to be able to continue both. The processes p_0 and p_1 should both communicate safely with each other. We support `_⊕_` and `_&_` without the need to extend our notion of session. As a binary choice is no more than a single bit of information, offering a choice is receiving one bit and making a choice is sending one bit. The bit must be analysed to define the the rest of the session, using the eliminator for $\mathbb{2}$. If one thinks of sending (`!`) to be a Σ-type and of receiving (`?`) to be a Π-type, then the standard interpretation of $\&$ as a product $\times$ and of $\oplus$ as a coproduct $\uplus$ is recovered through these two type isomorphisms: $A \times B \cong \Pi(x : \mathbb{2}).\ \mathbb{2}\texttt{-elim}\ A\ B\ x$ and $A \uplus B \cong \Sigma(x : \mathbb{2}).\ \mathbb{2}\texttt{-elim}\ A\ B\ x$. Figure 4 contains the definitions for choices ($\&$ and $\oplus$), their introduction form, and their units.

The unit for $\&$ has to be equivalent to $\mathbb{1}$ (the unit for $\times$) and the unit for $\oplus$ has to be equivalent to $\mathbb{0}$ (the unit for $\uplus$). Another way of viewing this is that the send and receive actions (`!`, `?`) are *n*-ary choices and the units are of arity zero. Therefore the unit for $\&$ is defined as a receive in the empty type. The common pattern here is that messages are eliminated right away after a `recv`, a `?`, and a `!`. As a particular case the $\texttt{in}\top$, after having received a value in the empty type, can pretend to follow any other session S. This cannot happen anyway as an eliminator only reduces once it is given a concrete value, which is impossible in the case of $\mathbb{0}$. Send and receive actions can therefore be used to recover the additive fragment of linear logic. Choices of any arity are available by choosing the message type to be of a particular size. The elimination of the message then permits as many continuations for the session as there are choices available.

These examples are thus defined for an extension of a system such as AGDA which would encompass all the features of ***PTT***. As notational convenience the system silently takes care of ended channels. Similarly, sessions are implicitly ended, e.g. by writing $?(x : \mathbb{N}).\ ?(y : \mathbb{N})$ without the final $.\ \perp$. Moreover multiple receives (or sends) on the same channel can be written in a single go, written as `recv c(x : ℕ)(y : ℕ)` instead of `recv c(x : ℕ) recv c(y : ℕ)`. The syntax for splitting channels (⅋ and ⊗), `c{d,e}` and `c[d,e]` respectively.

$$S_0 \mathbin{\&} S_1 = ?(x : \mathbb{2}).\, \mathbb{2}\text{-}\mathtt{elim}\, S_0\, S_1\, x$$

$$S_0 \oplus S_1 = !(x : \mathbb{2}).\, \mathbb{2}\text{-}\mathtt{elim}\, S_0\, S_1\, x$$

$$\frac{\Gamma \vDash P_0 :: [c : S_0], \mathcal{J} \quad \Gamma \vDash P_1 :: [c : S_1], \mathcal{J}}{\Gamma \vDash c(x : \mathbb{2})\ \mathbb{2}\text{-}\mathtt{elim}\, P_0\, P_1\, x :: [c : S_0 \mathbin{\&} S_1], \mathcal{J}}$$

$$\frac{\Gamma \vDash P :: [c : S_0], \mathcal{J}}{\Gamma \vDash \overline{c}\langle 0_2 \rangle\, P :: [c : S_0 \oplus S_1], \mathcal{J}}$$

$$\top = ?(x : \mathbb{0}).\, \mathbb{0}\text{-}\mathtt{elim}\, x$$

$$0 = !(x : \mathbb{0}).\, \mathbb{0}\text{-}\mathtt{elim}\, x$$

$$\frac{}{\Gamma \vDash c(x : \mathbb{0})\ \mathbb{0}\text{-}\mathtt{elim}\, x :: [c : \top], \mathcal{J}}$$

```
S_0 & S_1 = ?(x : 𝟚). 𝟚-elim S_0 S_1 x

S_0 ⊕ S_1 = !(x : 𝟚). 𝟚-elim S_0 S_1 x

in& : ⦃ S_0; J ⦄ × ⦃ S_1; J ⦄ → ⦃ S_0 & S_1; J ⦄
in& ⟨ p_0 , p_1 ⟩ =
  π(c;ds). recv c(x : 𝟚) (𝟚-elim p_0 p_1 x)@(c;ds)

in⊕ : ⦃ S_0; J ⦄ ⊎ ⦃ S_1; J ⦄ → ⦃ S_0 ⊕ S_1; J ⦄
in⊕ (inl p) = π(c;ds). send c < 0_2 > p@(c;ds)
in⊕ (inr p) = π(c;ds). send c < 1_2 > p@(c;ds)

⊤-Session = ?(x : 𝟘). 𝟘-elim x

0-Session = !(x : 𝟘). 𝟘-elim x

in⊤ : ⦃ ⊤-Session; J ⦄
in⊤ = π(c;ds). recv c(x : 𝟘) (𝟘-elim x)@(c;ds)
```

Figure 4 Deriving the additive connectives and units

Identity expansion as forwarders. Previous correspondences between linear logic and the π-calculus have put a process forwarder as the interpretation of the identity rule. [4, 19]. One might wonder why ***PTT*** does not have any identity rule, nor any builtin forwarder process. The reason being that such a process can be expressed in our language. This process is defined by induction over the type of sessions similarly to how one proves identity expansion in linear logic. For both send and receive, there is one side on which one can receive, from this side we receive a message that is then forwarded to the other side, and recursively forward the continuation of the session. For $\parr$ and $\otimes$ one splits both channels, recursively forwarding the two continuations in parallel. In our extended AGDA syntax it reads as follows:

```
fwd : Π(S : Session_0). ⦃ S ; S^⊥ ⦄
fwd(?(x:A). S x) = π(i;o). recv i(x : A) send o <x> fwd(S x)@(i;o)
fwd(!(x:A). S x) = π(o;i). recv i(x : A) send o <x> fwd(S x)@(o;i)
fwd(S_0 ⅋ S_1) = π(i;o). i{i_0,i_1} o[o_0,o_1] ( fwd(S_0)@(i_0;o_0) | fwd(S_1)@(i_1;o_1) )
fwd(S_0 ⊗ S_1) = π(o;i). i{i_0,i_1} o[o_0,o_1] ( fwd(S_0)@(o_0;i_0) | fwd(S_1)@(o_1;i_1) )
```

Distributed merge-sort. As a way to present and demonstrate the expressiveness of our system we define the essential building blocks and hint at various possible enhancements. A distributed merge-sort usually relies on a traditional sequential algorithm to be used once the size of the vector is small enough. We are going assume a type for vectors called `Vec A n` where `A` is the type of the elements and `n` is its size. This is a data type commonly featured in the dependently typed programming languages. Moreover we assume a type `Ord A` which encompasses a comparison function and potentially a proof that it is a total ordering on `A`. On vectors we assume the following functions which can all be defined in a type theory such as AGDA:

```
take  : Π(A : Set)(m : ℕ)(n : ℕ)(v : Vec A (m + n)). Vec A m
drop  : Π(A : Set)(m : ℕ)(n : ℕ)(v : Vec A (m + n)). Vec A n
merge : Π(m : ℕ)(n : ℕ)(v_0 : Vec ℕ m)(v_1 : Vec ℕ n). Vec ℕ (m + n)
sort  : Π(A : Set)(ord : Ord A)(n : ℕ)(v : Vec A n). Vec A n
ord-ℕ : Ord ℕ
```

We define `Sorter A n` as `?(vi : Vec A n). !(vo : Vec A n)`, that is, a session which receives a vector and sends back a vector of the same size. Then our first sorting process is receiving the input vector and sending back the sorted vector using the function `sort`.

```
simple-sorter : Π(n : ℕ). ⦃ Sorter ℕ n ⦄
simple-sorter n = π(c). recv c(vi : Vec ℕ n) send c < sort ℕ ord-ℕ n vi >
```

A way to enhance the sorting session is to enforce that the resulting vector is the sorted version of the input vector. To capture this, we define the type `Sorted n v = Σ(v' : Vec ℕ n). v' ≡ sort n v`, and refine the protocol as follows:

```
precise-sorter : Π(n : ℕ). ⦃ ?(vi : Vec ℕ n). !(vo : Sorted n vi) ⦄
precise-sorter n = π(c). recv c(vi : Vec ℕ n) send c < ⟨ sort ℕ ord-ℕ n vi , refl ⟩ >
```

So far the sorting process works only on vectors of natural numbers with the predefined total ordering `ord-ℕ`. One can scale to the general case by first sending the type `A` of elements, `ord` the total ordering on `A`, the length `n`, and finally the input vector. This process shows some key aspects such as the ability to send types, functions (`ord`) and to reference them later in the session. Due to the treatment of universe levels the resulting session is a $\mathtt{Session}_1$:

```
polymorphic-sorter : ⦃ ?(A : Set_0). ?(ord : Ord A). ?(n : ℕ). Sorter A n ⦄
polymorphic-sorter = π(c). recv c(A : Set)(ord : Ord A)(n : ℕ)(vi : Vec A n)
                     send c < sort A ord n vi >
```

We now turn to the distributed merge sort. We implement a merger process which communicates with two sorters both implementing the `Sorter` session for size `m` and `n`. Our merger can then implement `Sorter` for the size `m + n`. It first splits the only channel as two sub-channels `d` and `io`, the channel `d` is then split again to have access to the two individual helper processes, then it receives the input vector on `io`, sends both halves to d_0 and d_1, then receives the sorted sub-vectors again from d_0 and d_1, and finally it sends the merge vectors back on channel `io`. The whole session is thus essentially three times the `Sorter` session combined with ⅋, which gives the right amount of control to the merging process. Finally we put a wrapper around the merger where the channels are packaged with ⅋ and the result is nicely presented with the use of linear implication ($A \multimap B = A^{\perp} ⅋ B$):

```
merger  : Π(m : ℕ)(n : ℕ). ⦃ (Sorter ℕ m)^⊥ ; (Sorter ℕ n)^⊥ ; Sorter ℕ (m + n) ⦄
merger m n = π(e_0;e_1;io).
  recv io(vi : Vec ℕ (m + n)) send e_0 < take ℕ m n vi > send e_1 < drop ℕ m n vi >
  recv e_0(v_0 : Vec ℕ m) recv e_1(v_1 : Vec ℕ n) send io < merge m n v_0 v_1 >

merger-⊸ : Π(m : ℕ)(n : ℕ). ⦃ (Sorter ℕ m ⊗ Sorter ℕ n) ⊸ Sorter ℕ (m + n) ⦄
merger-⊸ m n = π(c). c{d,io} d{e_0,e_1} merger m n @(e_0;e_1;io)
```

The merge sort function can then combine the previous processes together to sort the vectors. The function first checks whether it is worthwhile to distribute the work further. If it is not, the `simple-sorter` is used, otherwise the size is divided in two, two channels are created, and three sub-processes are launched. The first two are recursive calls and each of them uses a different channel, the third one is the merger which uses the two other end points from the two helpers and the channel `c` from upstream:

```
merge-sort : Π(n : ℕ). ⦃ Sorter ℕ n ⦄
merge-sort n
  | n < 100000 = simple-sorter n
  | otherwise  = π(io). (ν d_0 e_0) (ν d_1 e_1)
   ( merge-sort n_0 @(d_0) | merge-sort n_1 @(d_1) | merger n_0 n_1 @(e_0;e_1;io) )
   where ⟨ n_0 , n_1 ⟩ = n /2
```

Remark that one might wonder if this merge program is too sequential: we cannot add any more parallelism since this would inevitably violate one of the typing rules. However, there is a form of parallelism in this program, in the sense that actions on distinct channels can commute, and thus the first action ready to interact can be considered first.

6 Conclusion and Future Work

The question of extending programming or reasoning frameworks with elements of concurrent programming, or linearity [5, 13] has been studied in various ways: here, we have seen that the burden of supporting linearity inside type theory is limited when considering concurrency and linear types as a mere extension, wrapped with care into a functional layer.

The key to the approach of ***PTT*** is to limit the scope while offering simple means of specifying programs exploiting both communication primitives and dependent types. Further investigations should establish the limits to reasonable specifications, to know in particular what kind of code mobility can be represented by sending and receiving processes wrapped as terms, or to which extent session types themselves can be communicated and used to make processes even more dependent. Note that in ***PTT***, such advanced use will always use complex entanglements of the functional and concurrent layers.

Another question is that of representations: frameworks based on functional programming are good at reasoning on sequential languages, but offer no natural support for reasoning about concurrency. If enough complex examples can be easily expressed inside a system following the methodology of ***PTT***, the task of reasoning about processes could be greatly simplified. But so far such proofs are primarily intrinsic, as opposed to specifications as predicates on processes. One first step would be to study the consequences of including the congruence relation on processes into the conversion relation, thus enabling some reasoning on the equality of processes. Another promising direction is concerned with information coming out of processes. Indeed, if a process has type $\{\!|\, !(x : A) \,|\!\}$, its normal form is necessarily a send and thus holds a value of type A. We can generalise this by observing that processes following sessions composed only of $!$ and $\parr$ can be extracted as nested Σ-types. By introducing an operation on sessions written $\texttt{log}\ S$, which unlike the duality operator converts only $\otimes$ to $\parr$ and $?$ to $!$, we can for instance extract the transcript of a communication between processes. This can be expressed in the style of a cut, interpreted as a linear formula, using $(S \otimes S^{\perp}) \multimap \texttt{log}\ S$.

Finally, on a more practical level, the development of a language integrating concurrency on top of Agda would give the opportunity to exploit the features of this system and make it easier to specify and verify practical implementations of concurrent algorithms.

References

1 Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

2 Arnon Avron. The method of hypersequents in the proof theory of propositional non-classical logics. In *Logic: from foundations to applications*, pages 1–32. 1996.

3 Gianluigi Bellin and Philip Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

4 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236, 2010.

5 Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002.

6 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

7 Simon Gay and Vasco Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

8 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

9 Jean-Yves Girard. Proof-nets : the parallel syntax for proof-theory. In A. Ursini and P. Agliano, editors, *Logic and Algebra*. M. Dekker, New York, 1996.

10 Alessio Guglielmi. A system of interaction and structure. *Transactions on Computational Logic (ACM)*, 8(1):1–64, 2007.

11 Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

12 Kohei Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523, 1993.

13 Neelakantan Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *POPL'15*, pages 17–30, 2015.

14 Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

15 Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

16 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

17 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M. Felleisen and P. Gardner, editors, *ESOP'13*, volume 7792 of *LNCS*, pages 350–369, 2013.

18 Vasco Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

19 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.