A Dependently-Typed Linear π -Calculus in Agda

(Article begins on next page)

A Dependently-Typed Linear π-Calculus in Agda

Luca Ciccone
luca.ciccone@unito.it
Università di Torino

Luca Padovani
luca.padovani@unito.it
Università di Torino

ABSTRACT

Session types have consolidated as a formalism for the specification and static enforcement of communication protocols. Many different theories of dependent session types have been proposed, some enabling refined specifications on the content of messages, others allowing the structure of the protocols to depend on data exchanged in the protocol itself. In this work we continue a line of research studying the foundations of binary session types. In particular, we propose a variant of the linear π-calculus whose type structure encompasses virtually all dependent session types using just two type constructors: linear channel types and linear dependent pairs. We use Agda not only to formalize the metatheory of the calculus and obtain machine-checked proofs of type soundness, but also as host language in which we implement data-dependent protocols.

CCS CONCEPTS

• **Theory of computation → Process calculi**; **Type structures**; *Program specifications*; *Program analysis*.

KEYWORDS

linear π-calculus, dependent session types, binary sessions, Agda

ACM Reference Format:
Luca Ciccone and Luca Padovani. 2018. A Dependently-Typed Linear π-Calculus in Agda. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Session type systems [17, 18] can statically and modularly guarantee the absence of communication errors in well-typed programs. Every session type system revolves around the following key ideas. First, it associates each *endpoint* of a communication channel – or *session* – with a *session type* that specifies type and direction of messages flowing through that endpoint. Second, the type system makes sure that the session types associated with the endpoints of a session describe complementary protocols so that an input/output action performed on one of the endpoints is matched by a corresponding counteraction on the other endpoint. This relation between session types is often called *duality*. Finally, the type system checks that the sequence of actions performed by a process on an endpoint matches with the session type of the endpoint. To do so, it forbids simultaneous access to the same endpoint by concurrent processes. As a consequence, session endpoints are treated as *linearized* resources that can only be accessed by a single process at any given time, although their ownership can be passed around through a mechanism called *delegation*.

Considering the relevant applications of dependent types in the description of *data formats* [10, 12, 22] and of *protocols* [3, 4, 11], it is not surprising that they have drawn the attention of the session type community as well, where they have led to a variety of different dependent session type languages. For example, Toninho et al. [31] and Griffith and Gunter [16] use dependent session types to provide refined specifications on the content of exchanged messages. Toninho and Yoshida [32] study a dependent session type theory where the structure of both types and processes may depend on the content of messages. Thiemann and Vasconcelos [30] propose a label-dependent variant of session types in which the ability to describe branching protocols does not require dedicated type constructors or communication primitives.

Except for the work of Thiemann and Vasconcelos, all the other dependent session type systems follow a common pattern, by *extending* a non-dependent session type system with new or refined constructs to express predicates on messages and/or dependencies between messages and behaviors. Thiemann and Vasconcelos [30] recognize that the increased expressiveness enabled by dependent types can in fact be exploited to *streamline* the structure of session types and *reduce* the number of type constructors that are necessary to describe complex protocols. Our work aims at taking their result one step further, by proposing a deconstruction of dependent session types in terms of an arguably minimal set of orthogonal features: *linear channels* and *linear dependent pairs*. We argue that these two ingredients suffice to encode the structure of virtually every known dependent session type for *binary* sessions.

The inspiration for our work comes from the *encoding* of binary sessions into the linear π-calculus [20] thoroughly studied by Dardha et al. [6] and based on the following idea: a sequence of communications occurring on a session endpoint can be encoded as a sequence of one-shot communications on a chain of *linear channels*. Unlike session endpoints, which can be used repeatedly – albeit sequentially – for several communications, a linear channel can be used only once. To model a long, structured conversation using linear channels, the sender of a message *pairs* the payload with a *continuation*, that is a new linear channel from which the rest of the conversation proceeds. Even from this informal description it is clear that (non-dependent) pairs play a major role in the encoding studied by Dardha et al. [6]. Our main insight is that, by promoting non-dependent pairs to dependent ones, not only we simplify some aspects of their encoding, but we also encompass a broader range of protocol specifications that includes dependent session types.

There are both theoretical and practical motivations that make these encodings worth investigating. On the theoretical side, they

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

deepen our understanding of the fundamental principles underlying session types. For example, they allow us to draw useful connections between notions – think of *subtyping* – that may look counterintuitive when defined on session types directly [13, 14], but that are recognizable as folklore if explained in terms of their encoding [6]. On the practical side, Padovani [24, 25] has shown that working with encoded session types may enable session type checking and inference in general-purpose programming languages having no specific support for session types. Our work sets the stage for a light implementation of dependent session types in Agda.

Summary of contributions. We define a dependently-typed version of the linear π-calculus dubbed DLπ in which *the structure of communications may depend on the content of exchanged messages in a strong sense.* DLπ is stratified in a *process layer* used to model communications and a *functional layer* used to compute not only messages, but also data-dependent processes and types.

In line with some previous works [16, 31], we only describe the process layer of DLπ leaving its functional layer mostly unspecified. To substantiate our results, we provide a *complete Agda formalization of DLπ's metatheory and machine-checked proofs of type soundness.* A novel aspect of this formalization is that *we intertwine DLπ and Agda so that we can write data-dependent processes and types taking full advantage of Agda's features.*

Finally, *we describe the systematic encoding of some representative session type languages* [17, 30, 31] into DLπ's types. This way, we extend the results of Dardha et al. [6] to dependent session types using a minimal, unifying model that encompasses a variety of dependent session type systems.

Structure of the paper. We describe syntax and semantics of DLπ in Section 2 and illustrate it through a series of examples in Section 3. Although the examples are inspired to the encoding of binary sessions in the linear π-calculus, they address scenarios in which the structure of the protocol strongly depends on exchanged messages. The dependent type system of DLπ is described in Section 4, where we also formulate the main properties of well-typed processes. Section 5 provides a bird's eye view of the Agda formalization of DLπ, focusing on the definition of the key data types. We also show the Agda implementation of some of the examples discussed in Section 3. In Section 6 we show the encoding into DLπ's types of three representative session type languages. We discuss related work in more detail in Section 7 and conclude in Section 8. Appendix A provides additional examples and Agda code that could not be accommodated in the main part of the paper. The full Agda formalization of DLπ and of the encodings described in Section 6 can be downloaded from a public repository [5].

2 SYNTAX AND SEMANTICS

As anticipated in Section 1, DLπ consists of a *functional layer* in which we express *computations* and a *process layer* in which we express *communications*. We do not detail the function layer, from which we inherit a set $\mathcal{M}$ of *pure terms*, ranged over by p and q. We assume that $\mathcal{M}$ includes the *unit value* `tt`, the booleans true and false, the natural numbers and that it is closed by pair construction.

The syntax of the process layer is shown in Table 1. We make use of infinite sets $\mathcal{C}$ and $\mathcal{X}$ of *channels* and *variables* and we call

Domains	$a, b, c \in \mathcal{C}$	channels
	$x, y, z \in \mathcal{X}$	variables
	$u, v \in \mathcal{C} \cup \mathcal{X}$	names
	$p, q \in \mathcal{M}$	pure terms
Terms	$M, N ::= p$	pure term
	$\mid\ u$	name
	$\mid\ M, N$	pair
Processes	$P, Q ::= \mathsf{idle}$	inaction
	$\mid\ u(x).P$	input
	$\mid\ u\langle M\rangle$	output
	$\mid\ \mathsf{let}\ x, y = M\ \mathsf{in}\ P$	pair splitting
	$\mid\ P \mid Q$	parallel composition
	$\mid\ (a)P$	restriction
	$\mid\ {*}P$	replication

Table 1: Syntax of DLπ.

names channels and variables without distinction. DLπ terms consist of variables, pure terms, channels and pairs. Any pure term can seamlessly flow from the functional layer to the process layer, where it can be used for constructing messages. The flow of terms from the process layer to the functional layer is also possible and useful, but it requires more care. We illustrate it in Section 3 and discuss the necessary precautions in Section 4.

By and large, DLπ processes are like π-calculus processes. We write idle for the inactive process that performs no action. A process of the form $u(x).P$ waits for a message x from channel u and then continues as P. A process of the form $u\langle M\rangle$ sends a message M on channel u. For the sake of simplicity, we only consider asynchronous communications, in which the output operation is not followed by a continuation. Synchronous communication does not pose substantial problems and is left out just because it is not used in this work. Parallel composition, replication and name restriction are standard. A process of the form $\mathsf{let}\ x, y = M\ \mathsf{in}\ P$ inspects the value M, which is supposed to be a pair, and then continues as P where the first and second component of the pair have been respectively bound to x and y. It is often the case that pair splitting immediately follows an input prefix. For this reason, we define $u(x, y).P$ as syntactic sugar for $u(z).\mathsf{let}\ x, y = z\ \mathsf{in}\ P$ for some z that does not occur elsewhere.

The notions of free and bound names for expressions and processes are standard, bearing in mind that the only binders are input prefixes, channel restrictions and pair splitting. We write $\mathsf{fn}(P)$ for the set of free names occurring in P.

One key aspect that is not self-evident from the syntax of DLπ is that we use the functional layer not only as a language for expressing terms, but also as a language for computing processes and types. In other words, any occurrence of the meta-variables P and Q in Table 1 may stand for an expression of the functional layer that computes a process. This mechanism allows us to compute processes from data and is not fundamentally different from monadic I/O as found in Haskell or Agda, where pure functions can

Structural congruence $\boxed{P \equiv Q}$

S-PAR-IDLE $\mathsf{idle} \mid P \equiv P$

S-PAR-COMM $P \mid Q \equiv Q \mid P$

S-PAR-ASSOC $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

S-NEW-IDLE $(a)\mathsf{idle} \equiv \mathsf{idle}$

S-NEW-COMM $(a)(b)P \equiv (b)(a)P$

S-PAR-NEW $$\frac{a \notin \mathsf{fn}(Q)}{(a)P \mid Q \equiv (a)(P \mid Q)}$$

S-PAR-REP $*P \equiv P \mid *P$

Reduction $\boxed{P \xrightarrow{\alpha} Q}$

R-COM $a\langle M\rangle \mid a(x).P \xrightarrow{a} P\{M/x\}$

R-STRUCT $$\frac{P \equiv\xrightarrow{\alpha}\equiv Q}{P \xrightarrow{\alpha} Q}$$

R-LET $\mathsf{let}\ x, y = M, N\ \mathsf{in}\ P \xrightarrow{\tau} P\{M, N/x, y\}$

R-NEW $$\frac{P \xrightarrow{\alpha} Q \qquad \alpha \neq c}{(c)P \xrightarrow{\alpha} (c)Q}$$

R-TAU $$\frac{P \xrightarrow{c} Q}{(c)P \xrightarrow{\tau} (c)Q}$$

R-PAR $$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}$$

Table 2: Operational semantics of DLπ.

be used used to compute I/O actions with side effects [26]. We will see examples of data-dependent processes starting from Section 3.

The operational semantics of DLπ is given in Table 2 in terms of a *structural congruence* relation $\equiv$ and a reduction relation $\xrightarrow{\alpha}$.

Structural congruence is standard. Axioms S-PAR-IDLE, S-PAR-COMM and S-PAR-ASSOC express commutativity and associativity of parallel composition, idle acting as the identity. Axiom S-NEW-IDLE removes/introduces unused channels, S-NEW-COMM swaps restrictions and S-PAR-NEW expands/shrinks the scope of a restricted channel. Finally, S-PAR-REP captures the standard meaning of a replication $*P$ as an unbounded availability of P's.

The reduction relation is labelled by *actions* which are either channels or the special label τ indicating an internal computation step. The label is necessary only to formulate and prove the subject reduction result (Theorem 4.3) and has no other operational relevance.

Rule R-COM is the synchronization between an output $a\langle M\rangle$ and an input prefix $a(x).P$ on the same channel a, whereby x is replaced by M in the continuation of the input prefix. We write $P\{M/x\}$ for the capture-avoiding substitution of M for the free occurrences of x in P. Rule R-LET formalizes the semantics of splitting a pair M, N and substituting M and N for x and y in the continuation.

The remaining rules close reduction under parallel composition (R-PAR), structural congruence (R-STRUCT) and restrictions (R-NEW and R-TAU). In R-TAU, a synchronization on channel c turns into an internal reduction when crossing the restriction of c.

3 STRUCTURED CONVERSATIONS

We illustrate how to represent structured conversations in DLπ through a series of examples that are directly inspired to the encoding of binary sessions into the linear π-calculus [6].

3.1 Linear conversations

In a typical session calculus [17], a process like

$$u!1.u?(x).print!x \tag{3.1}$$

describes a structured conversation where the session endpoint u is first used to send a number and then to receive a response x, which is then forwarded on another channel *print*. A peer process that communicates successfully with this one is

$$u?(x).u!\langle x + 1\rangle \tag{3.2}$$

which receives a number x from the session endpoint u and sends its successor back on the same endpoint.

In general, the *same* session channel can be used for exchanging an arbitrary number of messages. In sharp contrast, a linear channel can be used for a single communication only, after which the channel is depleted and cannot be used again. To encode a structured communication using linear channels, we pair the payload with a fresh channel which is used for the subsequent communication. For example, in DLπ we write (3.1) as $P_1(u)$ where

$$P_1(u) \stackrel{\text{def}}{=} (c)(u\langle 1, c\rangle \mid c(x).print\langle x\rangle) \tag{3.3}$$

Before the payload 1 is sent on the linear channel u, a fresh channel c is created and sent along with the payload. That is the channel from which the response is expected. The reader may be worried by the fact that the two communications which were performed in sequence in (3.1) have become parallel activities in (3.3). The reason why the flow of information – and therefore the structure of the conversation – is the same in (3.1) and (3.3) is that the input operation on c performed by $P_1(u)$ can only be completed once the message $u\langle 1, c\rangle$ has been delivered, since that message is the only place in which the other peer (to be discussed in a moment) will find a reference to the new channel c. The use of fresh (linear) channels is key in preserving the structure of the conversation.

Let us now look at the DLπ encoding of (3.2):

$$Q_1(u) \stackrel{\text{def}}{=} u(x, y).y\langle x + 1\rangle \tag{3.4}$$

This process waits for a pair x, y from u and then sends the successor of x on y. In this case the server *does not* create a fresh channel to pair with the result $x + 1$, since no further communications are expected after this one. In general, though, the same idea of creating fresh channels we have discussed for the client may also apply here, if the conversation is supposed to continue.

3.2 Conversations with message predicates

As a simple variation of the scenario discussed in Section 3.1, consider a conversation in which a client process sends a natural number n to a server process that computes its predecessor. Since the predecessor function is only defined for strictly positive natural numbers, the client is expected to send evidence of the fact that $n > 0$ along with the number itself. For the sake of illustration, we model this conversation so that the number n and the evidence $\mathsf{nzero}(n)$ are transmitted as *two subsequent messages*. For the time being, we ignore the details about the structure of the term $\mathsf{nzero}(n)$ and assume that properties like this one can be expressed in the functional layer. In Section 5 we will see a concrete realization of this example in Agda.

We can model the client process thus

$$P_2(u) \stackrel{\text{def}}{=} (b)(u\langle 2, b\rangle \mid (c)(b\langle \mathsf{nzero}(2), c\rangle \mid c(x).\mathit{print}\langle x\rangle))$$

where we observe that each message sent to the server is paired with a fresh channel used for the subsequent communication. As before, we use parallel composition and rely on fresh channels to ensure that the communications occur in the expected order, so that the server receives $\mathsf{nzero}(2)$ only after 2 has already been received. The server is defined as

$$Q_2(u) \stackrel{\text{def}}{=} u(x, v).v(y, w).w\langle \mathsf{pred}(x, y)\rangle \tag{3.5}$$

where the function $\mathsf{pred}(x, y)$ computes $x - 1$ given a proof y that $x > 0$. Note that y states a property concerning a message x received earlier. At the type level, this will translate into the fact that x, v is a *dependent* pair (see Section 4.7).

3.3 Conversations with branching points

A structured conversation may proceed along different paths depending on a *choice* taken by one of the interacting processes. A choice taken autonomously by a process but not communicated to the peer would almost certainly result in chaos. For this reason, any choice that affects the structure of subsequent communications must be encoded and communicated in some form, for example as a boolean value or as a label chosen from a known set.

In "traditional" session calculi – those not making use of dependent types – the dependency between this boolean value or label and the session type that describes the rest of the conversation is handled by a dedicated construct precisely because the type system would otherwise be unable to express this dependency. Thiemann and Vasconcelos [30] observe that, in presence of richer types, such dependency falls within the expressiveness of the type system without requiring ad hoc constructs.

Consider a scenario in which a server is able to provide both behaviors described earlier by $Q_1(u)$ and $Q_2(u)$. The server first waits for a label that identifies the desired behavior. In this case, a plain boolean value suffices to discriminate between the two possibilities. After that, the server performs the operation corresponding to the received label. We can model a persistent server with this capabilities as the process

$$Q_3 \stackrel{\text{def}}{=} *a(x, y).F(x, y) \tag{3.6}$$

where F is a function (expressed in the functional layer) that, applied to a boolean argument x and a channel y, yields a process according to the following equations:

$$\begin{aligned} F(\mathsf{true}\,, y) &= Q_1(y) \\ F(\mathsf{false}, y) &= Q_2(y) \end{aligned}$$

Recall that the conversations carried out by $Q_1(y)$ and $Q_2(y)$ have different structures (and different lengths). At the type level, these differences translate into the fact that y has different types in $Q_1(y)$ and $Q_2(y)$. Such differences can be reconciled if the pair x, y received by Q_3 in (3.6) is in fact a *dependent pair*, so that the type of y depends on the value of x.

A client of Q_3 first sends a boolean value indicating which operation it requests and then behaves accordingly. For example,

$$P_{31} \stackrel{\text{def}}{=} (c)(a\langle \mathsf{true}, c\rangle \mid P_1(c))$$

accomplishes the same task as P_1 when composed with Q_3 and

$$P_{32} \stackrel{\text{def}}{=} (c)(a\langle \mathsf{false}, c\rangle \mid P_2(c))$$

accomplishes the same task as P_2. Considering that Q_3 is a persistent, the composition $P_{31} \mid P_{32} \mid Q_3$ satisfies both clients with no interferences from each other, despite the fact that they request different operations with different conversation structures.

3.4 Variable-length conversations

Another common instance of conversation with data-dependent structure is the exchange of a sequence of messages whose length depends on some previous message. As an example, think of a server that receives a number n and computes the product of the n subsequent messages. We can model such server as the process

$$Q_4 \stackrel{\text{def}}{=} *a(n, v).F(n, v, 1) \tag{3.7}$$

where F is the function defined by the following equations

$$\begin{aligned} F(0 \quad\;\;, v, z) &= v\langle z\rangle \\ F(n + 1, v, z) &= v(x, y).F(n, y, x * z) \end{aligned}$$

whose third argument z is used as accumulator for the result. When $n = 0$, the channel v is used for sending back the result in the accumulator. When $n > 0$, the channel v is used for receiving a number x in the sequence along with another channel y that will be used for the next communication.

A client that interacts with (3.7) to compute the factorial of a number n could be defined thus:

$$P_4 \stackrel{\text{def}}{=} (c)(a\langle n, c\rangle \mid G(n, c))$$

where G is the function defined by the following equations:

$$\begin{aligned} G(0 \quad\;\;, v) &= v(x).\mathit{print}\langle x\rangle \\ G(n + 1, v) &= (c)(v\langle n + 1, c\rangle \mid G(n, c)) \end{aligned}$$

Note once again how $G(n, v)$ uses v differently – for one input or for one output – depending on whether $n = 0$ or not.

4 TYPE SYSTEM

4.1 Multiplicities

We use the *multiplicities* 0, 1 and ω for keeping track – approximately – of the number of times a channel is used according to a given input/output capability. Specifically, 0 means that a channel is never used, 1 that it is used exactly once and ω that it is used an arbitrary number of times, possibly never. We define two operations $+$ and $\cdot$ to "combine" and "scale" multiplicities, thus:

$$\begin{array}{ccc} 0 + \sigma = \sigma + 0 = \sigma & 1 + 1 = \omega & \omega + \sigma = \sigma + \omega = \omega \\ 0 \cdot \sigma = \sigma \cdot 0 = 0 & 1 \cdot \sigma = \sigma \cdot 1 = \sigma & \omega \cdot \omega = \omega \end{array}$$

Note that $(\{0, 1, \omega\}, +, \cdot)$ is a commutative semiring. When no confusion may arise, we abbreviate $\sigma \cdot \rho$ as $\sigma\rho$.

Domains	$A, B \in \mathcal{A}$	pure types
	$\sigma, \rho \in \{0, 1, \omega\}$	multiplicities
Types	$t, s ::= A$	pure type
	$\mid\ {}^{\sigma,\rho}[t]$	channel type
	$\mid\ \Sigma(x : t)s$	linear dependent pair

Table 3: Syntax of types.

4.2 Types

Types are ranged over by t and s and their syntax is given in Table 3. We inherit from the functional layer a set $\mathcal{A}$ of *pure types* ranged over by A and B. We assume that $\mathcal{A}$ includes types such as $\top$ (the unit type with just one constructor `tt`), `Bool`, $\mathbb{N}$ and dependent pairs $\Sigma(x : A)B$ as well. A *channel type* has the form ${}^{\sigma,\rho}[t]$ and describes a channel that is used σ times for receiving *and* ρ times for sending messages of type t. A *linear dependent pair type* has the form $\Sigma(x : t)s$ and describes pairs whose first component has type t and whose second component has type s. Since the variable x is bound in s, the type of the second component may depend on the value of the first one in a way that will be made more precise in Section 4.3. The "linear" qualification means that pairs having this type can only be used *once*. In the following we write $t \times s$ for *linear non-dependent pair types*, which are the degenerate case of $\Sigma(x : t)s$ when x does not occur in s.

We extend the operations $+$ and $\cdot$ defined on multiplicities to types in the following way:

$$A + A = A \qquad\qquad \sigma \cdot A = A$$
$${}^{\sigma_1,\rho_1}[t] + {}^{\sigma_2,\rho_2}[t] = {}^{\sigma_1+\sigma_2,\rho_1+\rho_2}[t] \qquad \sigma \cdot {}^{\rho_1,\rho_2}[t] = {}^{\sigma\rho_1,\sigma\rho_2}[t]$$

Intuitively, a type $t + s$ cumulates the uses of a channel that is used according to t in some part of a program and according to s in some other part of the same program. For example, the equation

$${}^{1,0}[t] + {}^{0,1}[t] = {}^{1,1}[t]$$

captures the fact that a channel that is used somewhere for receiving a message of type t and somewhere else for sending a message of type t is used once for sending and once for receiving a message of type t overall. The equation holds precisely because the sentence sounds like a tautology.

The operation $\sigma \cdot t$ yields the type of a resource of type t that is used σ times. For example, the equivalence

$$\omega \cdot {}^{1,0}[t] = {}^{\omega,0}[t]$$

captures the fact that using zero or more times a channel from which a single message of type t is received is the same as using the channel for receiving zero or more messages of type t. Note that neither type combination nor type scaling affect the type of messages exchanged through channels.

Unlike the operations $+$ and $\cdot$ on multiplicities (Section 4.1), the operations $+$ and $\cdot$ on types are *partial*: neither is defined on linear dependent pairs and $+$ is undefined when combining types having different shapes. Also, two channel types can be combined with $+$ only if they are used for exchanging messages of the *same type*.

We say that a type is unrestricted if it describes a resource that can be discarded or used an arbitrary number of times and we say that a type is linear otherwise. We can make this distinction precise in terms of idempotency of $+$, thus:

Definition 4.1 (unrestricted and linear types). A type t is *unrestricted* if $t = t + t$ and it is *linear* otherwise.

All pure types are unrestricted, just like channels types whose multiplicities are 0 or ω. Note that channels with an unrestricted type can be used for exchanging messages whose type is linear. The type ${}^{1,1}[t]$ is linear, since a channel with this type must be used once for sending and once for receiving a message of type t. In general, a channel type may specify different constraints on the number of uses for each capability. For example, a channel with type ${}^{\omega,1}[t]$ is used an unspecified number of times for receving messages of type t, but it is used only once for sending a message of type t. Since $+$ is undefined on linear dependent pairs, such pairs are strictly linear resources that must be used once, either by sending them in a message or by splitting them with a `let`.

All types of the form $0{\cdot}t$ and $\omega{\cdot}t$ are unrestricted. We occasionally use these forms of scaling to enforce the fact that certain types are unrestricted.

4.3 More on dependent pairs

Linear dependent pairs $\Sigma(x : t)s$ belong to the process layer and are not to be confused with pure dependent pairs $\Sigma(x : A)B$ in the functional layer. While the latter are a special case of the former, the dependency expressed in linear dependent pairs is somewhat unconventional. Let us see why.

We have said that every pure term is also a DLπ term (Section 2) and that every pure type is also a DLπ type (Table 3). This flow of terms and types from the functional layer to the process layer allows us to take advantage of all the features provided by the functional layer in the modeling and typing of processes. The flow of terms in the other direction, from the process layer to the functional layer, is also useful and doubly so. First, that is the mechanism we have used in Section 3 for *computing processes from messages*. Second, we use the same mechanism also for *computing types from messages*. Nonetheless, there is a fundamental distinction between these two uses of DLπ terms in the functional layer. When a DLπ term is used to compute a process, our type system is able to track the uses of the resources occurring in the term (most notably, channels) by looking at the result of the computation. But in a dependent pair $\Sigma(x : t)s$, where a DLπ term x may occur within a type s, we lose control on whether and how x is used. In fact, we argue that it makes no sense to consider a type s that depends on the *identity* of channels possibly occurring in x.

To prevent these issues, we *filter* DLπ terms that are used in types through a map $[\![\cdot]\!]$ that "erases" all the channels occurring in them. The map $[\![\cdot]\!]$ is defined thus

$$[\![p]\!] = p \qquad [\![x]\!] = x \qquad [\![a]\!] = \mathtt{tt} \qquad [\![M, N]\!] = [\![M]\!], [\![N]\!]$$

and identifies all channels with the uninformative value `tt`. The pure term $[\![M]\!]$ corresponding to M is the *view* of M as seen in the functional layer, from which everything but the channels in M can be accessed. The filtering on terms induces a filtering on types

$$[\![A]\!] = A \qquad [\![{}^{\sigma,\rho}[t]]\!] = \top \qquad [\![\Sigma(x : t)s]\!] = \Sigma(x : [\![t]\!])[\![s]\!]$$

that obliterates all channel types from a DLπ type.

We can now refine the informal description of linear dependent pairs given earlier. A type $\Sigma(x : t)s$ describes those pairs M, N such that M has type t and N has type $s\{[\![M]\!]/x\}$. This way, the channels possibly occurring in M are not duplicated as a result of the substitution and their identity cannot affect the type of N.

4.4 Contexts

We use contexts to track the type of free names occurring in processes and terms, hence to provide an abstract description of a process behavior in terms of the resources it uses. A *context* Γ is a finite, partial map from names to types written $u_1 : t_1, \dots, u_n : t_n$. We write $\emptyset$ for the empty context, $\mathrm{dom}(\Gamma)$ for the domain of Γ, namely for the (finite) set of names for which there is an association in Γ, we write $\Gamma(u)$ for the type associated with u in Γ when $u \in \mathrm{dom}(\Gamma)$ and Γ, Δ for the union of Γ and Δ when $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$.

We need to combine and scale contexts, pretty much like we need to combine and scale types. Intuitively, the combination $\Gamma + \Delta$ accounts for the cumulated use of resources by two processes, one described by Γ and the other described by Δ. Context composition is the partial operation defined by the following equations:

$$
\begin{aligned}
\Gamma + \Delta &= \Gamma, \Delta && \text{if } \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset \\
(u : t, \Gamma) + (u : s, \Delta) &= (u : t + s), (\Gamma + \Delta)
\end{aligned}
$$

Note that $\Gamma + \Delta$ is defined provided that all the names for which there is an association in both Γ and Δ have combinable types. In this case, we have $\mathrm{dom}(\Gamma + \Delta) = \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta)$.

The scaling of Γ with respect to σ, written $\sigma \cdot \Gamma$, provides an abstract description of σ copies of a process described by Γ. Context scaling is the partial operation defined by the equations

$$
\begin{aligned}
\sigma \cdot \emptyset &= \emptyset \\
\sigma \cdot (u : t, \Gamma) &= (u : \sigma \cdot t), (\sigma \cdot \Gamma)
\end{aligned}
$$

provided that every type in the range of Γ can be scaled by σ.

We extend to contexts the terminology introduced in Definition 4.1 for types. Specifically, we say that Γ is unrestricted if so are all the types in its range. All contexts of the form $0 \cdot \Gamma$ and $\omega \cdot \Gamma$ are unrestricted.

As we have discussed in Section 4.3, resources available in the process layer should also be available in the functional layer of DLπ, albeit in a filtered form. For this reason, the typing judgments of DLπ will refer to a pair of contexts $\Psi; \Gamma$ respectively describing the resources available in the functional and those available in the process layer. We say that Ψ is a *pure context* and we require Ψ to *agree* with Γ in the following sense: every resource $u \in \mathrm{dom}(\Gamma)$ with type $\Gamma(u)$ available in the process layer is also available with type $\Psi(u) = [\![\Gamma(u)]\!]$ in the functional layer. In general, Ψ may describe more resources than those described by Γ. This can happen for two reasons. First, the functional layer may provide resources – such as library functions, built-in data types, etc. – that are not defined within processes but that are nonetheless essential for building and computing processes. Second, it could be the case that a linear resource (e.g., a pair) contains data that is needed in the functional layer, and yet the resource is not visible in Γ because it is already used by another part of the process. In these cases, Ψ will contain associations for linear resources that are in scope but not in $\mathrm{dom}(\Gamma)$ (see the discussion on T-PAR later on).

Typing rules for terms $\boxed{\Psi; \Gamma \vdash M : t}$

$$
\text{T-PURE}\ \frac{\Psi \vdash p : A}{\Psi; 0 \cdot \Gamma \vdash p : A}
\qquad
\text{T-NAME}\ \Psi; 0 \cdot \Gamma, u : t \vdash u : t
$$

$$
\text{T-PAIR}\ \frac{\Psi; \Gamma \vdash M : t \qquad \Psi; \Delta \vdash N : s\{[\![M]\!]/x\}}{\Psi; \Gamma + \Delta \vdash M, N : \Sigma(x : t)s}
$$

Typing rules for processes $\boxed{\Psi; \Gamma \vdash P}$

$$
\text{T-IDLE}\ \Psi; 0 \cdot \Gamma \vdash \mathsf{idle}
\qquad
\text{T-INPUT}\ \frac{\Psi; \Gamma \vdash u : {}^{1,0}[t] \qquad \Psi, x : [\![t]\!]; \Delta, x : t \vdash P}{\Psi; \Gamma + \Delta \vdash u(x).P}
$$

$$
\text{T-OUTPUT}\ \frac{\Psi; \Gamma \vdash u : {}^{0,1}[t] \qquad \Psi; \Delta \vdash M : t}{\Psi; \Gamma + \Delta \vdash u\langle M \rangle}
$$

$$
\text{T-LET}\ \frac{\Psi; \Gamma \vdash M : \Sigma(x : t)s \qquad \Psi, x : [\![t]\!], y : [\![s]\!]; \Delta, x : t, y : s \vdash P}{\Psi; \Gamma + \Delta \vdash \mathsf{let}\ x, y = M\ \mathsf{in}\ P}
$$

$$
\text{T-PAR}\ \frac{\Psi; \Gamma \vdash P \qquad \Psi; \Delta \vdash Q}{\Psi; \Gamma + \Delta \vdash P \mid Q}
\qquad
\text{T-NEW}\ \frac{\Psi; \Gamma, c : {}^{\sigma,\rho}[t] \vdash P}{\Psi; \Gamma \vdash (c)P}
\qquad
\text{T-REP}\ \frac{\Psi; \Gamma \vdash P}{\Psi; \omega \cdot \Gamma \vdash *P}
$$

Table 4: Typing rules.

4.5 Typing rules

The typing rules for expressions and processes are shown in Table 4. The former ones derive judgments of the form $\Psi; \Gamma \vdash M : t$ meaning that M is well typed in $\Psi; \Gamma$ and has type t. The latter ones derive judgments of the form $\Psi; \Gamma \vdash P$ meaning that P is well typed in $\Psi; \Gamma$. In both cases, we make the implicit assumption that Ψ agrees with Γ. We now describe the rules in detail.

Axiom T-PURE lifts a well-typed term in the functional layer to the process layer. We do not detail how judgments $\Psi \vdash p : A$ are derived, as they depend on the functional layer. Notice that the context in which p is well typed has the form $0 \cdot \Gamma$, hence it is unrestricted, recording the fact that p does not use any linear resource.

Axiom T-NAME states that a name u (that is, a variable or a channel) is well typed and has type t in a context that contains an association $u : t$. The unused part of the context must have the form $0 \cdot \Gamma$, recording the fact that no resource apart from u is used.

Rule T-PAIR states that if M has type t and N has type s in which x is a placeholder for M, then the pair M, N has type $\Sigma(x : t)s$. There are two twists that set this rule apart from a conventional introduction for dependent pairs. The first one is that we replace $[\![M]\!]$ – and *not* M – for x in s, as we have discussed in Section 4.3. In addition, the contexts Γ and Δ used for typing M and N are *combined* in the conclusion of the rule, so as to cumulate the uses of resources that occur in both M and N. As an example, if the same channel occurs in M with type ${}^{0,1}[t]$ and also in N with type ${}^{1,0}[t]$, then it occurs in the pair M, N with type ${}^{1,1}[t]$.

Now on to the typing rules for processes. Axiom T-IDLE states that, since the idle process performs no action and uses no resource, it is well typed in an unrestricted context of the form $0 \cdot \Gamma$.

Rule T-INPUT states that an input processes of the form $u(x).P$ is well typed if the name u has type ${}^{1,0}[t]$ and the continuation P is well typed in a context enriched with the association $x : t$. In the continuation, the pure context is enriched with the association $x : [\![t]\!]$ so that the pure part of the received message x is available in the functional layer as well. The type of u indicates that u is a channel used (here) for a single input operation of a message of type t. The whole process is well typed in a context that combines the resources used in both u and P.

Rule T-OUTPUT deals with processes of the form $u\langle M\rangle$. The name u must have a type of the form ${}^{0,1}[t]$, indicating that it is a channel used (here) for a single output of a message M of type t. As we have seen in other rules, the contexts used for typing u and M are combined in the conclusion.

Rule T-LET deals with processes of the form $\texttt{let}\ x, y = M\ \texttt{in}\ P$. The term M must have a type of the form $\Sigma(x : t)s$ and P must be well typed in a context enriched with the associations for x and y. Similarly to what we have seen in T-INPUT, the pure context is also enriched with associations for the same names, although their types are the filtered versions of t and s.

Rules T-PAR and T-NEW handle parallel compositions and channel restrictions in the expected way. T-PAR illustrates better than other rules why Ψ may contain associations for resources that are in scope but not visible in the context. A linear resource u used by P will have an association in Γ but not in Δ. Yet, process Q may refer to non-linear components of u through the pure context Ψ, which is the same for P and Q. In T-NEW, we do not constrain in any way the multiplicities σ and ρ occurring in the type of a restricted channel, even though some combinations of σ and ρ may indicate obvious flaws in the process. For example, restricted channels with type ${}^{0,1}[t]$ or ${}^{1,0}[t]$ or ${}^{0,\omega}[t]$ suggest the presence of unmatched input or output operations and may cause deadlocks or yield orphan messages. We ignore such issues in this paper since our type system is not aimed at enforcing progress or other liveness properties [23]. Note also that, in T-NEW, the pure context is *not* enriched with an association for the channel c. Since channels are mapped to the constant `tt` in the functional layer, there is no need to augment the pure context in this case.

Rule T-REP deals with replicated processes of the form $*P$. Since a replicated process P is morally equivalent to an unbounded number of copies of P running in parallel, the rule scales the context in which P is well typed by ω. As a side effect, such context cannot contain pairs, for which scaling is undefined.

4.6 Properties of well-typed processes

We summarize here the main properties of well-typed processes, starting from the fact that structural congruence preserves typing.

THEOREM 4.2. *If* $\Psi; \Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Psi; \Gamma \vdash Q$.

To formulate the property that typing is preserved also by reductions, we have to consider that the type associated with a channel may change as the result of a communication taking place on that channel. In particular, a linear channel can be used for a single communication only. For this reason, the reduct Q of a process P after a communication is well typed in a context that is related to – though not necessarily the same as – the context in which P is well typed. We express this relationship between contexts through a relation $\xrightarrow{\alpha}$ defined by the following two axioms:

$$\Gamma \xrightarrow{\tau} \Gamma \qquad \Gamma + c : {}^{1,1}[t] \xrightarrow{c} \Gamma$$

Unobservable actions do not change the context. A communication on a channel c is allowed provided that the channel is associated with a channel type in which neither multiplicity is 0. The type of the channel in the resulting context is suitably adjusted to account for this communication. In particular, we have

$$c : {}^{1,1}[t] \xrightarrow{c} c : {}^{0,0}[t]$$

capturing the fact that a linear channel c is "consumed" and no longer usable after a communication takes place on c. Subject reduction can now be formulated showing that all the reductions in processes are simulated by matching reductions in contexts:

THEOREM 4.3 (SUBJECT REDUCTION). *If* $\Psi; \Gamma \vdash P$ *and* $P \xrightarrow{\alpha} Q$, *then there exists* Δ *such that* $\Gamma \xrightarrow{\alpha} \Delta$ *and* $\Psi; \Delta \vdash Q$.

The converse of Theorem 4.3, in which every reduction in a context can be simulated by the process, does not hold in general since the structure of the process may constrain the order in which communications take place.

We now discuss a few safety properties guaranteed by the type system, most of which are in fact corollaries of Theorem 4.3. First of all, we can formulate *communication safety* as the property that a message received from a channel has the expected type.

PROPOSITION 4.4. *If* $\Psi; \Gamma \vdash u\langle M\rangle \mid u(x).P$, *then there exist* t, Γ_1 *and* Γ_2 *such that* $\Psi; \Gamma_1 \vdash M : t$ *and* $\Psi; \Gamma_2, x : t \vdash P$.

The next two results specifically concern linearity. The first one states that a name that is not used by a well-typed process P – that is, a name not occurring free in P – must have an unrestricted type. In other words, the type system ensures that names with linear types are not discarded without first being used.

PROPOSITION 4.5. *If* $\Psi; \Gamma, u : t \vdash P$ *and* $u \notin \mathsf{fn}(P)$, *then* $t = 0 \cdot t$.

The second result states that a channel on which a communication occurs has non-zero multiplicities in its type. In other words, the type system ensures that channels whose type has 0 multiplicities are not used for communications.

PROPOSITION 4.6. *If* $\Psi; \Gamma, c : {}^{\sigma,\rho}[t] \vdash P$ *where* $\sigma\rho = 0$ *and* $P \xrightarrow{\alpha} Q$, *then* $\alpha \neq c$.

Other properties of well-typed processes that hold for the linear π-calculus [20], including race-freedom and partial confluence for communications on linear channels, hold in DLπ too.

4.7 Examples

To give a flavor of the type system, we sketch the typing derivations for the processes Q_1 and Q_2 in Section 3. To reduce clutter, we omit from the judgments the pure context Ψ, whose precise content is inessential and can be partially guessed from the presented elements. Take $t_1 \stackrel{\text{def}}{=} {}^{1,0}[s]$ where $s \stackrel{\text{def}}{=} \mathbb{N} \times {}^{0,1}[\mathbb{N}]$. The proof tree below shows that Q_1 is well typed in the context $u : t_1$:

$$\dfrac{u : t_1 \vdash u : t_1 \qquad \dfrac{z : s \vdash z : s \qquad \dfrac{y : {}^{0,1}[\mathbb{N}] \vdash y : {}^{0,1}[\mathbb{N}] \qquad \dfrac{\text{functional layer}}{x : \mathbb{N} \vdash x + 1 : \mathbb{N}}}{x : \mathbb{N}, y : {}^{0,1}[\mathbb{N}] \vdash y\langle x + 1\rangle}}{z : s \vdash \mathsf{let}\ x, y = z\ \mathsf{in}\ y\langle x + 1\rangle}}{u : t_1 \vdash u(z).\mathsf{let}\ x, y = z\ \mathsf{in}\ y\langle x + 1\rangle}$$

Note how the contexts are split so as to distribute the resources where they are needed to type the process. The details of the typing derivation for the judgment $x : \mathbb{N} \vdash x + 1 : \mathbb{N}$ depend on the type system of the functional layer and are omitted.

We now show that the process Q_2 in (3.5) is well typed in the context $u : t_2$ where

$$t_2 \stackrel{\text{def}}{=} {}^{1,0}[\Sigma(x : \mathbb{N})\ {}^{1,0}[(x > 0) \times {}^{0,1}[\mathbb{N}]]]$$

The dependent pair allows us to relate the number x received from the channel and the proof that $x > 0$, which is received from a *different linear channel*. We have the following derivation, in which we have elided the applications of T-NAME and collapsed the applications of T-IN immediately followed by T-LET:

$$\dfrac{\begin{array}{c}\vdots\\ u\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ v\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ z\end{array} \qquad \dfrac{\text{functional layer}}{x : \mathbb{N}, y : x > 0 \vdash \mathsf{pred}(x, y) : \mathbb{N}}}{x : \mathbb{N}, y : x > 0, z : {}^{0,1}[\mathbb{N}] \vdash z\langle \mathsf{pred}(x, y)\rangle}}{x : \mathbb{N}, v : s \vdash v(y, z).z\langle \mathsf{pred}(x, y)\rangle}}{u : t \vdash u(x, v).v(y, z).z\langle \mathsf{pred}(x, y)\rangle}$$

We postpone the typing derivations for the processes Q_3 in (3.6) and Q_4 in (3.7) to the end of Section 5, where we will be able to show them in full using Agda for computing processes and types.

5 AGDA FORMALIZATION

In this section we sketch an embedding of DLπ in Agda. We use Agda not just as a tool for formalizing the metatheory of DLπ, but also as a particular instantiation of its functional layer. This way, we can rely on a full-fledged, dependently-typed language for computing processes and types. Space constraints force us to discuss a slightly simplified version of the formalization and to focus on the definition of the Agda data types we use for representing types, contexts and processes. The rest of the formalization follows in a fairly straightforward way once these data types are in place. The full development is available in a public repository [5].

We begin with multiplicities, represented as a Mult data type with 3 constructors corresponding to the elements 0, 1 and ω.

```
data Mult : Set where
  #0 #1 #ω : Mult
```

Even though the operations + and · on multiplicities are easy to implement as Agda functions, we find it more convenient to express them as relations, for two different reasons. First, combination and scaling are only partially defined for types and contexts, hence they must be expressed as – or with the help of – relations for those entities anyway. Using relations also for multiplicities allows us to give a uniform presentation of these operations on all the entities for which they make sense. In addition to that, multiplicities occur in DLπ types, which in turn occur in contexts, which in turn occur in the Agda type of terms and processes. Having functions that compute indexes would force us to adopt heterogeneous notions of equality that, in our experience, often result in unreasonably complex Agda code. For this reason, we prefer working with relations wherever possible.

We define two data types

```
data MScale : Mult → Mult → Set
data MSplit : Mult → Mult → Mult → Set
```

in such a way that MScale σ ρ is inhabited if and only if $\rho = \omega\sigma$ and MSplit σ σ_1 σ_2 is inhabited if and only if $\sigma = \sigma_1 + \sigma_2$. Note that the typing rules shown in Table 4 scale contexts, types and multiplicities by 0 and ω, whereas with MScale we only consider scaling by ω. It is simpler to provide ad hoc unary predicates to expresses the properties $t = 0 \cdot t$ and $\Gamma = 0 \cdot \Gamma$.

The Agda data type for representing DLπ types is an inductive-recursive definition [9], since it refers to the $[\![\cdot]\!]$ function that maps DLπ types into the corresponding pure types. We have

```
mutual
  data Type : Set1 where
    Pure  : Set → Type
    Chan : Mult → Mult → Type → Type
    Pair  : (t : Type) → (⟦ t ⟧ → Type) → Type

  ⟦_⟧ : Type → Set
  ⟦ Pure A ⟧       = A
  ⟦ Chan _ _ _ ⟧ = ⊤
  ⟦ Pair t f ⟧      = ∑ ⟦ t ⟧ λ x → ⟦ f x ⟧
```

where the constructors of Type correspond to the three forms of DLπ types (Table 3) and $[\![\cdot]\!]$ is the filtering function defined in Section 4.3 that maps DLπ types into Agda types. Note that Type is in Set$_1$ since its Pure constructor has a Set argument. In the full development [5], Type is actually a sized type [1] and the Type argument of Chan is a thunk to account for possibly infinite types.

We provide TScale and TSplit relations to express scaling and splitting of types, along with a predicate TNull t that holds if and only if $0 \cdot t = t$.

```
data TNull  : Type → Set1
data TScale : (t s : Type) → ⟦ t ⟧ → ⟦ s ⟧ → Set1
data TSplit  : (t t1 t2 : Type) → ⟦ t ⟧ → ⟦ t1 ⟧ → ⟦ t2 ⟧ → Set1
```

The TScale and TSplit data types have way more indexes than one would reasonably expect. As we will see shortly, whenever we scale or split types, we are always in the position of saying which pure term has its type being scaled or splitted. The additional indexes in TScale and TSplit allow us to "cast" the pure term to the types resulting from the scaling or the splitting. For example, an Agda value of type TScale t s p q witnesses that $s = \omega \cdot t$ and that if p is a pure term of type $[\![t]\!]$, then q is the corresponding pure term of type $[\![s]\!]$. Funnily enough, p and q are always *equal*, but their types $[\![t]\!]$ and $[\![s]\!]$ may differ. It would be trivial to write a casting function that, given a pure term of type $[\![t]\!]$, yields the same pure term with type $[\![s]\!]$. The problem, once again, is that using such casting function for computing indexes leads to unmanageable Agda code.

Following Benton et al. [2], Thiemann [29], Wadler and Kokke [33], we use intrinsically typed terms and processes where names are referenced through their de Brujin index. There is one key difference with these works, though, which allows us to intertwine the DLπ process layer and the Agda functional layer. Our contexts are not just lists of types, as in the aforementioned works, but rather lists of pairs $t \# p$ where t is a type and p is a pure term of type $[\![t]\!]$. These pure terms make sure that a context Γ, which is used for typing DLπ terms and processes, *agrees* with the pure context Ψ known by Agda, in the sense of Section 4.4. To better understand the relevance of these pure terms, recall from Section 4.3 that the filtering function $[\![\cdot]\!]$ on DLπ terms is defined so that $[\![x]\!] = x$. The x on the left-hand side of this equation is a DLπ variable, whereas the x on the right-hand side is an Agda variable. By storing a pair $t \# x$ in a context, we associate a DLπ variable – which is represented namelessy by the position of the pair in the context – not only with its type t but also with its corresponding Agda variable x.

```
data Context : Set₁ where
  []      : Context
  _#_::_ : (t : Type) → ⟦ t ⟧ → Context → Context
```

The CNull, CScale and CSplit data types play for contexts the same roles that TNull, TScale and TSplit play for types. Specifically, the predicate CNull Γ requires each type in Γ to satisfy TNull:

```
data CNull : Context → Set₁ where
  []   : CNull []
  _::_ : ∀{ t p Γ } → TNull t → CNull Γ → CNull (t # p :: Γ)
```

For CSplit $\Gamma\ \Gamma_1\ \Gamma_2$, we provide constructors for either splitting an entry of Γ according to TSplit or moving it into Γ_1 through the L constructor or into Γ_2 through the R constructor:

```
data CSplit : Context → Context → Context → Set₁ where
  []   : CSplit [] [] []
  _::_ : ∀{ t t₁ t₂ p p₁ p₂ Γ Γ₁ Γ₂ } →
         TSplit t t₁ t₂ p p₁ p₂ → CSplit Γ Γ₁ Γ₂ →
         CSplit (t # p :: Γ) (t₁ # p₁ :: Γ₁) (t₂ # p₂ :: Γ₂)
  L_   : ∀{ Γ Γ₁ Γ₂ t p } → CSplit Γ Γ₁ Γ₂ →
         CSplit (t # p :: Γ) (t # p :: Γ₁) Γ₂
  R_   : ∀{ Γ Γ₁ Γ₂ t p } → CSplit Γ Γ₁ Γ₂ →
         CSplit (t # p :: Γ) Γ₁ (t # p :: Γ₂)
```

We now turn our attention to the representation of terms, starting from names. A DLπ name is an Agda value of type Name $k\ \Gamma\ t\ p$ where k is de Brujin index of the name, t is its type in the context Γ, and p is the pure term associated with the name. Keeping track of the de Brujin index of a name in its Agda type is useful to infer that two values of type Name actually refer to the *same* name. For example, in the statement of Proposition 4.4 we have two occurrences of u which are used in two different ways, for sending and for receiving a message. When we invert T-PAR, T-INPUT and T-OUTPUT we can "only" infer that these occurrences of u correspond to Agda values of type Name $k\ \Gamma\ {}^{0,1}[t]$ tt and Name $k\ \Delta\ {}^{1,0}[s]$ tt respectively. We use the knowledge that the two names have the same index k to prove $t = s$ and therefore that Proposition 4.4 holds.

The Name data type is defined thus:

```
data Name : ℕ → Context → (t : Type) → ⟦ t ⟧ → Set₁ where
  here : ∀{ Γ t p } → CNull Γ → Name zero (t # p :: Γ) t p
  next : ∀{ k Γ t s p q } → TNull s → Name k Γ t p →
         Name (suc k) (s # q :: Γ) t p
```

The here constructor corresponds to the first name in a context, so its index is 0. The next constructor corresponds to a name found at position k in the remainder of the context, so its index is $k + 1$. In both cases, the part of the context not concerning the name must satisfy the condition $0 \cdot \Gamma = \Gamma$, as required by T-NAME.

A DLπ term is an Agda value of type Term $\Gamma\ t\ p$, where Γ is the context in which the term is well typed, t is its type and p is the corresponding pure term.

```
data Term : Context → (t : Type) → ⟦ t ⟧ → Set₁ where
  name : ∀{ k Γ t p } → Name k Γ t p → Term Γ t p
  pure : ∀{ Γ A } → CNull Γ → (p : A) → Term Γ (Pure A) p
  pair : ∀{ Γ Γ₁ Γ₂ t f p q } → CSplit Γ Γ₁ Γ₂ →
         Term Γ₁ t p → Term Γ₂ (f p) q → Term Γ (Pair t f) (p , q)
```

The constructors relate to the forms of DLπ terms (Table 1) and their arguments match the premises of the typing rules (Table 4). A pair requires two sub-terms respectively typed in Γ_1 and Γ_2 that combine into Γ, as by the CSplit $\Gamma\ \Gamma_1\ \Gamma_2$ argument. At last we can appreciate the role of the pure term p attached to the Agda type of terms, which is used here for computing the type $(f\ p)$ of the second component of the pair.

A DLπ process is an Agda value of type Process Γ, where Γ is the context in which the process is well typed:

```
data Process : Context → Set₁ where
  Idle : ∀{ Γ } → CNull Γ → Process Γ
  Send : ∀{ Γ Γ₁ Γ₂ t p } → CSplit Γ Γ₁ Γ₂ →
         Term Γ₁ (Chan #0 #1 t) _ → Term Γ₂ t p → Process Γ
  Recv : ∀{ Γ Γ₁ Γ₂ t } → CSplit Γ Γ₁ Γ₂ →
         Term Γ₁ (Chan #1 #0 t) _ →
         ((x : ⟦ t ⟧) → Process (t # x :: Γ₂)) → Process Γ
  Let  : ∀{ Γ Γ₁ Γ₂ t f p q } → CSplit Γ Γ₁ Γ₂ →
         Term Γ₁ (Pair t f) (p , q) →
         ((x : ⟦ t ⟧) (y : ⟦ f x ⟧) → Process (t # x :: f x # y :: Γ₂))
         → Process Γ
  Par  : ∀{ Γ Γ₁ Γ₂ } → CSplit Γ Γ₁ Γ₂ →
         Process Γ₁ → Process Γ₂ → Process Γ
  New  : ∀{ Γ σ ρ t } → Process (Chan σ ρ t # _ :: Γ) → Process Γ
  Rep  : ∀{ Γ Δ } → CScale Γ Δ → Process Γ → Process Δ
```

By now, most elements of this definition are self-explanatory. The only feature that differs from the syntax of the calculus and from the typing rules is the last argument in the Recv and Let constructors, which is not just a value of type Process but is actually a function that computes such value from one or two arguments x and y, the variables being bound by the input prefix or by the let form. As their type suggests, these arguments represent the pure terms corresponding to the DLπ terms bound to x and y and are used to populate the context of the continuation process. In the Let constructor, note how the type $f\ x$ of the second component of the pair is computed from the Agda variable corresponding to the first component.

The full Agda development [5] formalizes the whole semantics of DLπ, namely structural congruence ≡ and reduction $\xrightarrow{\alpha}$. As usual in an intrinsically-typed language, both relations incorporate the corresponding type-preservation results (Theorems 4.2 and 4.3), from which the other properties stated in Section 4.6 follow.

We now revisit some of the examples discussed in Section 3, showing how they are implemented in Agda. Recall that values of type Process Γ are intrinsically-typed processes, so the terms we see below actually correspond to typing derivations, not just to processes. Starting from Q_1 in (3.4), we have:

```
t₁ : Type
t₁ = Chan #1 #0 (Pair (Pure ℕ) λ _ → Chan #0 #1 (Pure ℕ))

Q₁ : Process (t₁ # _ ∷ [])
Q₁ = Recv (L [])    (name (here [])) λ _ →
     Let   (L [])    (name (here [])) λ x _ →
     Send (R L []) (name (here [])) (pure (P ∷ []) (x + 1))
```

We use the L and R constructors to split the typing context so as to distribute resources where they are needed. Note also the function argument of Let, which gives access to the message x being received. The P constructor is a value of type TNull ℕ, witnessing that the type of x is unrestricted. For Q_2 in (3.5) we have

```
t₂ : Type
t₂ = Chan #1 #0 (Pair (Pure ℕ) λ x →
     Chan #1 #0 (Pair (Pure (x ≢ 0)) λ _ → Chan #0 #1 (Pure ℕ)))

Q₂ : Process (t₂ # _ ∷ [])
Q₂ = Recv (L [])       (name (here [])) λ _ →
     Let   (L [])       (name (here [])) λ x _ →
     Recv (R L [])     (name (here [])) λ _ →
     Let   (L R [])     (name (here [])) λ y _ →
     Send (R L R []) (name (here [])) (pure (P ∷ P ∷ []) (pred x y))
  where
    pred : (x : ℕ) (y : x ≢ 0) → ℕ
    pred zero y    = ⊥-elim (y refl)
    pred (suc x) _ = x
```

where we use the function argument of the Pair constructor for specifying the type $x \not\equiv 0$ of the subsequent message.

In the case of Q_3 in (3.6), we have to pattern match on the received boolean value to compute both the type t_3 and the process. In the latter case, we must weaken the context in which Q_1 and Q_2 are typed, since these processes expect only one name in their context, whereas Q_3 provides two, the first of which is the boolean value which is used by neither Q_1 nor Q_2:

```
t₃ : Type
t₃ = Chan #ω #0 (Pair (Pure Bool) (λ b → if b then t₁ else t₂))

Q₃ : Process (t₃ # _ ∷ [])
Q₃ = Rep (chan sc1 sc0 ∷ []) (
     Recv (L []) (name (here [])) λ _ →
     Let   (L []) (name (here [])) λ { true  _ → weaken Q₁
                                     ; false _ → weaken Q₂ })
```

The term chan sc1 sc0 scales the input multiplicity of the channel used by the process from 1 to ω, to account for the fact that the process is replicated.

In order to construct Q_4 in (3.7), it is convenient to define an auxiliary function f such that f n describes the exchange of the n messages from the client to the server process and then the communication of the result from the server back to the client:

```
f : ℕ → Type
f zero    = Chan #0 #1 (Pure ℕ)
f (suc n) = Chan #1 #0 (Pair (Pure ℕ) λ _ → f n)
```

Now the server Q_4 in (3.7) can be implemented thus:

```
t₄ : Type
t₄ = Chan #ω #0 (Pair (Pure ℕ) f)

Q₄ : Process (t₄ # _ ∷ [])
Q₄ = Rep (chan sc1 sc0 ∷ []) (
     Recv (L []) (name (here [])) λ _ →
     Let   (L []) (name (here [])) λ n _ → weaken (F n 1))
  where
    F : (n : ℕ) → ∀{ p } → ℕ → Process (f n # p ∷ [])
    F zero     z = Send (L []) (name (here [])) (pure [] z)
    F (suc n) z = Recv (L []) (name (here [])) λ _ →
                  Let   (L []) (name (here [])) λ x _ →
                  weaken (F n (x ∗ z))
```

The interested reader will find the implementation of the client processes P_i in Appendix A. It is clear from these examples that writing even simple DLπ terms and processes in Agda is quite tedious. Preliminary results with an inference algorithm have shown that most of the CNull and CSplit witnesses can be automatically inferred as long as the programmer provides the type of bound channels and variables. We plan to finalize this algorithm in a future update of the Agda formalization.

6 ENCODING DEPENDENT SESSION TYPES

In Section 3 we have seen a few examples of structured conversations modeled in DLπ and in Sections 4 and 5 we have shown how these conversation can be described in DLπ's type language. In this section we take a more systematic approach to assess the expressiveness of DLπ's type language. We consider three representative session type languages [17, 30, 31] and define encoding functions to compile them all into the type language of DLπ. For the sake of uniformity, we make a few cosmetic adjustments to the syntax of session types presented in the aforementioned works while preserving their characterizing features. In all cases, we limit ourselves to finite session types with binary choices and branches, but our results extend easily to possibly infinite session types with arbitrarily labelled choices.

Session types *à la* Honda [17] have been presented in the first work on session types, their syntax is shown below:

$$\begin{aligned} T, S &::= \mathsf{end} \mid ?m.T \mid !m.T \mid T \,\&\, S \mid T \oplus S \\ m &::= A \mid T \end{aligned} \tag{6.1}$$

The type end describes endpoints that are not used anymore. Input $?m.T$ and output $!m.T$ describe session endpoints respectively used for receiving and sending a message of type m and then according to T. Hereafter, m ranges over pure types A and over session types themselves. Branches $T \,\&\, S$ and choices $T \oplus S$ describe session endpoints respectively used for receiving and sending a single bit of information and then according to either T or S accordingly. We will make the assumption that this information is encoded as a value of type Bool, with true being the value for selecting T and false being the value for selecting S. Although session types *à la* Honda are not explicitly presented as *dependent* session types, branches and choices are in fact a simple form of dependency whereby the type of the endpoint after the communication depends on the boolean value that is exchanged. This will be clear when we discuss the encoding of session types into DLπ's types.

All theories of session types rely on some notion of *duality* that plays a key role in the encodings we are about to discuss. The dual of a session type T, often denoted by $\overline{T}$, is the session type obtained from T by swapping inputs with outputs, choices with branches, and leaving message types and end unchanged. For example, we have $\overline{(!m.\mathsf{end}) \oplus \mathsf{end}} = (?m.\mathsf{end}) \,\&\, \mathsf{end}$.

Dardha et al. [6] have shown how to encode session types in (6.1) in terms of linear channels, non-dependent pairs and disjoint sums. Below we rephrase their encoding as a function $[\![\cdot]\!]$ that maps session types into DLπ's types, using dependent pairs to subsume both non-dependent pairs and disjoint sums:

$$
\begin{aligned}
[\![\mathsf{end}]\!] &= {}^{0,0}[\top] \\
[\![?m.T]\!] &= {}^{1,0}[[\![m]\!] \times [\![T]\!]] \\
[\![!m.T]\!] &= {}^{0,1}[[\![m]\!] \times [\![\overline{T}]\!]] \\
[\![T \,\&\, S]\!] &= {}^{1,0}[\Sigma(x : \mathsf{Bool})\ \mathsf{if}\ x\ \mathsf{then}\ [\![T]\!]\ \mathsf{else}\ [\![S]\!]] \\
[\![T \oplus S]\!] &= {}^{0,1}[\Sigma(x : \mathsf{Bool})\ \mathsf{if}\ x\ \mathsf{then}\ [\![\overline{T}]\!]\ \mathsf{else}\ [\![\overline{S}]\!]]
\end{aligned}
$$

The encoding of end yields an unusable channel with null multiplicities. The encoding of an input $?m.T$ or an output $!m.T$ yields a linear channel used for receiving or sending a non-dependent pair whose first component is the encoding of m and whose second component is another channel resulting from the encoding of T. The encoding of m yields either A or $[\![T]\!]$, according to the shape of m. Note that the type of the second component of the pair in the encoding of $!m.T$ is not $[\![T]\!]$ but rather $[\![\overline{T}]\!]$. The reason why duality is used here is that $[\![\overline{T}]\!]$ specifies how the second component of the pair is used by the *receiver* of the pair, as opposed to $[\![T]\!]$ which describes the behavior of the *sender* of the pair. The encoding of a branch $T \,\&\, S$ yields a linear channel used for receiving a dependent pair whose first component is a boolean value x and whose second component is (a term that reduces to) either $[\![T]\!]$ or $[\![S]\!]$ depending on the value of x. The if x then t else s on the right-hand side of the equations is to be interpreted as a pure term of the functional layer rather than a DLπ type constructor. The encoding of a choice $T \oplus S$ follows a similar pattern. As for the encoding of outputs, here too the encoded continuations are dualized.

Session types *à la* Toninho et al. [31] extend those shown in (6.1) with existential and universal quantifiers, one dual to the other:

$$T, S \ ::= \ \cdots \ \mid \ \forall x : A.T \ \mid \ \exists x : A.T \tag{6.2}$$

The $\forall$ and $\exists$ quantifiers respectively correspond to input and output operations that bind the exchanged message to a name that can be used in the rest of the session type for describing properties related to that message. Toninho et al. [31] consider for example

$$\forall x : \mathbb{N}.\forall u : (x > 0).\exists y : \mathbb{N}.\exists v : (y > 0).\mathsf{end}$$

which describes the behavior of a process that receives a natural number x and a proof u that $x > 0$ and sends back another natural number y along with a proof v that $y > 0$.

The encoding function $[\![\cdot]\!]$ can be extended with the equations

$$
\begin{aligned}
[\![\forall x : A.T]\!] &= {}^{1,0}[\Sigma(x : A)[\![T]\!]] \\
[\![\exists x : A.T]\!] &= {}^{0,1}[\Sigma(x : A)[\![\overline{T}]\!]]
\end{aligned}
$$

to account for quantifiers in the expected way, again dualizing the continuation session type for the output operation.

Thiemann and Vasconcelos [30] embrace the idea that branches and choices are forms of dependent types and propose a streamlined session type language that features input/output actions akin to quantified session types in (6.2) along with a case x of $\{T, S\}$ construct that *reduces* to either T or S depending on the value of x:

$$T, S \ ::= \ \mathsf{end} \ \mid \ ?x : m.T \ \mid \ !x : m.S \ \mid \ \mathsf{case}\ x\ \mathsf{of}\ \{T, S\} \tag{6.3}$$

As an example, the choice $T \oplus S$ in (6.1) can be expressed as $!x : \mathsf{Bool}.\mathsf{case}\ x\ \mathsf{of}\ \{T, S\}$. Also in this case the encoding is straightforward, with the case construct that naturally translates to a conditional expression in the functional layer:

$$
\begin{aligned}
[\![\mathsf{end}]\!] &= {}^{0,0}[\top] \\
[\![?x : m.T]\!] &= {}^{1,0}[\Sigma(x : [\![m]\!])[\![T]\!]] \\
[\![!x : m.T]\!] &= {}^{0,1}[\Sigma(x : [\![m]\!])[\![\overline{T}]\!]] \\
[\![\mathsf{case}\ x\ \mathsf{of}\ \{T, S\}]\!] &= \mathsf{if}\ x\ \mathsf{then}\ [\![T]\!]\ \mathsf{else}\ [\![S]\!]
\end{aligned}
$$

Observe that none of the presented encodings is *injective* if we consider DLπ types equals according to Agda's propositional equality. For example, we have $[\![?\mathsf{Bool}.T]\!] = [\![T \,\&\, T]\!]$ for session types *à la* Honda, $[\![?\mathsf{Bool}.T]\!] = [\![T \,\&\, T]\!] = [\![\forall x : \mathsf{Bool}.T]\!]$ for session types *à la* Toninho et al. [31] and

$$
\begin{aligned}
&[\![?x : \mathsf{Bool}.\mathsf{case}\ x\ \mathsf{of}\ \{?y : \mathbb{N}.T, ?y : \mathbb{N}.S\}]\!] \\
&\quad = [\![?x : \mathsf{Bool}.?y : \mathbb{N}.\mathsf{case}\ x\ \mathsf{of}\ \{T, S\}]\!]
\end{aligned}
$$

for session types *à la* Thiemann and Vasconcelos. This is not entirely surprising, since DLπ types describe communication protocols at a rather low level of abstraction, but it also highlights that the semantics of different constructs provided by these session type languages overlap to some extent. In contrast, the encoding of Dardha et al. [6], which relies on *different* low-level types for pairs and sums, has been shown injective by Padovani [24]. This property is useful for pretty-printing automatically inferred communication protocols [21].

Another intriguing aspect of these encoding functions is their interplay with duality. Duality plays a key role in all session type theories and yet it is surprisingly subtle to define correctly [15]. In part, this is because duality affects the *whole* structure of a session type. Whenever a session type language is extended with new forms, such as quantifiers in (6.2) or label case analysis in (6.3), duality must be suitably extended as well. However, as observed by Dardha et al. [6], duality turns into a much simpler relation when we consider encoded session types and, using an appropriate representation of channel types, it boils down to *type equality* [24]:

Proposition 6.1. *Let* $\overline{{}^{\sigma,\rho}[t]} \stackrel{\text{def}}{=} {}^{\rho,\sigma}[t]$. *Then* $[\![\overline{T}]\!] = \overline{[\![T]\!]}$.

This property says that the encoding of the dual of T differs from the encoding of T solely in the order of the multiplicities in the *top-level* channel type constructor of $\llbracket \overline{T} \rrbracket$. As a consequence, writing rich protocol specifications in encoded form in DLπ makes it easier to use all the expressive power of the underlying functional layer without worrying about duality. For example, one can mix-and-match constructs from (6.2) and (6.3) to express branching points that depend not on a label, but rather on a property between messages. As an example, the type

$$ {}^{1,0}[\Sigma(x : \mathbb{N})\ {}^{1,0}[\Sigma(y : \mathbb{N})\ \mathsf{if}\ x <^{\mathrm{b}} y\ \mathsf{then}\ t\ \mathsf{else}\ s]] \tag{6.4}$$

describes a channel which is used for reading two natural numbers x and y and then according to t or s depending on whether or not x is smaller than y ($<^{\mathrm{b}} : \mathbb{N} \to \mathbb{N} \to \mathsf{Bool}$ is the builtin less-than boolean function in Agda). The "dual" of (6.4) is described by a type which is basically the same, but with the topmost multiplicities swapped. Note that the protocol described by (6.4) can be expressed somehow using the session type languages in (6.2) or (6.3), but in both cases it must be patched so that the outcome of the comparison $x <^{\mathrm{b}} y$ is explicitly transmitted as a boolean value.

In [5] we give Agda formalizations of all the encodings $\llbracket \cdot \rrbracket$ in this section, each with a *decoder* $\llbracket \cdot \rrbracket$ that is shown to be the inverse of $\llbracket \cdot \rrbracket$ when session types are considered up to bisimilarity.

7 RELATED WORK

Dependent session types. The first theories of dependent session types are those of Toninho et al. [31] and Griffith and Gunter [16]. These works augment session types with binders, thus allowing for the specification of message predicates. Toninho and Yoshida [32] present a full calculus combining functions and processes in which the structure of both types and processes may depend on the content of messages, as in our work. In particular, their session types can describe a protocol such as (6.4) albeit with a more complex type structure compared to our own (Table 3). Unlike Toninho and Yoshida and aligning with Griffith and Gunter [16], Toninho et al. [31], we leave the functional layer of DLπ unspecified, but we contribute an Agda formalization of the calculus. Thiemann and Vasconcelos [30] propose a full model of functions and processes enabling a simplified form of dependency whereby the structure of types and processes may depend on labels and possibly natural numbers. They introduce a conditional context extension operator that prevents dependencies on linear values and plays a similar role of the filtering function $\llbracket \cdot \rrbracket$ that erases channels.

Zhou [35] describes the theory and implementation of a refinement session type system where the type of messages can be refined by predicates that specify their properties and relationships.

Dependent types for data formats and protocols. Oury and Swierstra [22] showcase the expressiveness of dependent types in describing cryptographic protocols and data formats. In particular, our Type data type with dependent pairs has been inspired by their definition of data formats using induction-recursion [9]. The works of Bhatti et al. [3] and Brady and Hammond [4] advocate the usefulness of dependent types in the definition of (Embedded) Domain Specific Languages (EDSLs) for the description of network protocols. In particular, they show how dependent types capture precisely the type of operations that change state-sensitive resources (e.g. sockets) and enable specifications of data-sensitive protocols (e.g. communication of checksums). Scalas et al. [28] use a blend of behavioral and dependent function types for the precise specification of actor-based programs.

Formalizations of session type systems. Thiemann [29] gives the first mechanized proof of a calculus of functions and sessions. His type system distinguishes between types and session types, but only non-dependent pairs are considered. de Muijnck-Hughes et al. [7] describe an Idris EDSL where dependent types enable reasoning on value dependencies between exchanged messages. Zalakain and Dardha [34] give another Agda formalization of the linear π-calculus. They focus exclusively on the process layer and only consider channel types, using typing with leftovers instead of context splitting as we do. While context splitting relates more closely with the model (Section 4) and other presentations of linear and session calculi, leftovers allow for simpler mechanizations. Rouvoet et al. [27] describe a technique inspired by separation logic to specify and verify in Agda interpreters using linear resources. Among the case studies they discuss is a linearly-typed lambda calculus with primitives for session communications.

Linear π-calculus. Our main source of inspiration is the work of Dardha et al. [6], which emphasizes the role of pairs in the encoding of sessions using linear channels. Dardha et al. show not only the encoding of session types (as we do in Section 6), but also the encoding of processes and prove an operational correspondence between session-typed processes and encoded ones. We think that all of these results extend to our calculus as well. The same encoding is also discussed in earlier works by Kobayashi [19] and Demangeon and Honda [8]. Padovani [24, 25] describes an OCaml library of binary sessions which blends static session type inference with dynamic linearity checking. Encoded session types makes it possible to rely exclusively on OCaml's type system.

8 CONCLUDING REMARKS

Linear channels combined with linear dependent pairs go a long way in describing structured conversations that depend on the *content* of messages in a strong sense. We have studied this combination in DLπ, a dependently-typed linear π-calculus that provides a unifying model for a variety of dependent session type systems.

We have used Agda not only as the language in which we formalize the metatheory of DLπ, but also as a particular instance of DLπ's functional layer from which we inherit the fundamental machinery related to dependent pairs. The interplay between Agda and the process layer of DLπ is mediated so as to prevent the flow of channels from the process layer to the functional layer. This mediation also prevents the specification of protocols that depend on the identity of channels.

The Agda formalization of DLπ can form the basis for a lightweight library implementation of dependent session types in Agda, along the lines of similar libraries for other functional languages [24, 25]. Although the amount of annotations required for the typing of processes appears intimidating (Section 5), preliminary results with an inference algorithm have shown that these annotations can be automatically synthesized in many cases. We plan to finalize these developments in the near future.

REFERENCES

[1] Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. In *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010 (EPiC Series)*, Ekaterina Komendantskaya, Ana Bove, and Milad Niqui (Eds.), Vol. 5. EasyChair, 18–32. https://arxiv.org/pdf/1012.4896.pdf

[2] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. 2012. Strongly Typed Term Representations in Coq. *J. Autom. Reasoning* 49, 2 (2012), 141–159. https://doi.org/10.1007/s10817-011-9219-0

[3] Saleem Bhatti, Edwin Brady, Kevin Hammond, and James McKinna. 2009. Domain Specific Languages (DSLs) for Network Protocols (Position Paper). In *29th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2009 Workshops), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 208–213. https://doi.org/10.1109/ICDCSW.2009.64

[4] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. *Fundam. Inform.* 102, 2 (2010), 145–176. https://doi.org/10.3233/FI-2010-303

[5] Luca Ciccone and Luca Padovani. 2020. `DependentLinearPi`. Università di Torino. Retrieved May 22, 2020 from https://gitlab.di.unito.it/luca.padovani/DependentLinearPi

[6] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

[7] Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. 2019. Value-Dependent Session Design in a Dependently Typed Language. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019 (EPTCS)*, Francisco Martins and Dominic Orchard (Eds.), Vol. 291. Open Publishing Association, 47–59. https://doi.org/10.4204/EPTCS.291.5

[8] Romain Demangeon and Kohei Honda. 2011. Full Abstraction in a Subtyped pi-Calculus with Linear Types. In *Proceedings of CONCUR'11 (LNCS 6901)*. Springer, 280–296.

[9] Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

[10] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2010. The next 700 data description languages. *J. ACM* 57, 2 (2010), 10:1–10:51. https://doi.org/10.1145/1667053.1667059

[11] Simon Fowler. 2014. *Verified Networking using Dependent Types*. University of St Andrews. http://simonjf.com/writing/bsc-dissertation.pdf BSc Dissertation.

[12] Simon Fowler and Edwin Brady. 2013. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 28-30, 2013*, Rinus Plasmeijer (Ed.). ACM, 49. https://doi.org/10.1145/2620678.2620683

[13] Simon J. Gay. 2016. Subtyping Supports Safe Session Substitution. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 95–108. https://doi.org/10.1007/978-3-319-30936-1_5

[14] Simon J. Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Inf.* 42, 2-3 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[15] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020 (EPTCS)*, Stephanie Balzer and Luca Padovani (Eds.), Vol. 314. Open Publishing Association, 23–33. https://doi.org/10.4204/EPTCS.314.3

[16] Dennis Griffith and Elsa L. Gunter. 2013. LiquidPi: Inferrable Dependent Session Types. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings (Lecture Notes in Computer Science)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.), Vol. 7871. Springer, 185–197. https://doi.org/10.1007/978-3-642-38088-4_13

[17] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science)*, Eike Best (Ed.), Vol. 715. Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

[18] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36. https://doi.org/10.1145/2873052

[19] Naoki Kobayashi. 2002. Type Systems for Concurrent Programs. In *10th Anniversary Colloquium of UNU/IIST (LNCS 2757)*. Springer, 439–453. Extended version available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf.

[20] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947. https://doi.org/10.1145/330249.330251

[21] Hernán Melgratti and Luca Padovani. 2017. *An OCaml Implementation of Binary Sessions*. River Publishers, 243–263. http://riverpublishers.com/downloadchapter.php?file=RP_9788793519817C11.pdf

[22] Nicolas Oury and Wouter Swierstra. 2008. The power of Pi. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 39–50. https://doi.org/10.1145/1411204.1411213

[23] Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 72:1–72:10. https://doi.org/10.1145/2603088.2603116

[24] Luca Padovani. 2017. A simple library implementation of binary sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

[25] Luca Padovani. 2019. Context-Free Session Type Inference. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 9:1–9:37. https://doi.org/10.1145/3229062

[26] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/

[27] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 284–298. https://doi.org/10.1145/3372885.3373818

[28] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 502–516. https://doi.org/10.1145/3314221.3322484

[29] Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 19:1–19:15. https://doi.org/10.1145/3354166.3354184

[30] Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-dependent session types. *Proc. ACM Program. Lang.* 4, POPL (2020), 67:1–67:29. https://doi.org/10.1145/3371135

[31] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, Peter Schneider-Kamp and Michael Hanus (Eds.). ACM, 161–172. https://doi.org/10.1145/2003476.2003499

[32] Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, 128–145. https://doi.org/10.1007/978-3-319-89366-2_7

[33] Philip Wadler and Wen Kokke. 2019. *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/.

[34] Uma Zalakain and Ornela Dardha. 2020. *Typing the linear pi calculus in Agda*. University of Glasgow. Retrieved May 25, 2020 from https://github.com/umazalakain/typing-linear-pi

[35] Fangyi Zhou. 2019. *Refinement Session Types*. Imperial College London. https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-ug-projects/ZhouF-Refinement-Session-Types.pdf BSc Dissertation.

A SUPPLEMENT

In this appendix we show the code corresponding to the client processes P_1–P_4 described in Section 3. In general these processes are slightly more intricate than the corresponding servers because they need to create new channels which must be suitably distributed among parallel sub-processes.

Since these processes behave in a dual manner with respect to the corresponding servers, it is useful to define a dual-of function that computes the "dual" of a DLπ type (see Proposition 6.1), so that we can reuse most of the structure already given in the types t_1–t_4 of Section 5.

```
dual-of : Type → Type
dual-of (Chan σ ρ t) = Chan ρ σ t
dual-of t            = t
```

Note that dual-of behaves as the identity on any type other than channel types. We add this case to dual-of just so that the function is total. Starting from P_1 in (3.3) we have

```
P1 : Process (dual-of t1 # _ :: [])
P1 = New (Par (chan sp01 sp10 :: L [])
              (Send (R L []) (name (here [])) (pair (R []) (pure [] 2) (name (here []))))
              (Recv (L []) (name (here [])) λ _ → Idle (P :: [])))
```

where sp$\sigma\rho$ is a witness for the relation MSplit ($\sigma + \rho$) σ ρ. Note that s_1 differs from t_1 (Section 5) solely for the topmost multiplicities. The same will apply for all the types used by the client processes presented hereafter. Concerning P_2 in Section 3.2 we have:

```
nzero : (n : ℕ) → suc n ≢ 0
nzero _ ()
```

```
P2 : Process (dual-of t2 # _ :: [])
P2 = New (Par (chan sp10 sp01 :: L [])
              (Send (R L []) (name (here [])) (pair (R []) (pure [] 2) (name (here []))))
              (New (Par (chan sp01 sp10 :: L [])
                        (Send (R L []) (name (here [])) (pair (R []) (pure [] (nzero 1)) (name (here []))))
                        (Recv (L []) (name (here [])) λ _ → Idle (P :: [])))))
```

For P_{31} and P_{32} in Section 3.3 we have:

```
P31 : Process (dual-of t3 # _ :: [])
P31 = New (Par (chan sp10 sp01 :: L [])
               (Send (R L []) (name (here []))
                     (pair (R []) (pure [] true) (name (here []))))
               P1)
```

```
P32 : Process (dual-of t3 # _ :: [])
P32 = New (Par (chan sp10 sp01 :: L [])
               (Send (R L []) (name (here []))
                     (pair (R []) (pure [] false) (name (here []))))
               P2)
```

Now the process composition at the end of Section 3.3 can be typed thus:

```
P3 : Process (Chan #0 #ω _ # _ :: [])
P3 = Par (chan sp00 sp11 :: []) P31 P32
```

```
C3 : Process (Chan #ω #ω _ # _ :: [])
C3 = Par (chan sp0ω spω0 :: []) P3 Q3
```

We conclude with the definition of P_4 in Section 3.4:

```
G : (n : ℕ) → ∀{ p } → Process (dual-of (f n) # p :: [])
G zero       = Recv (L []) (name (here [])) λ _ →
                 Idle (P :: [])
G (suc zero) = New (Par (chan sp01 sp10 :: L [])
                        (Send (R L []) (name (here [])) (pair (R []) (pure [] 1) (name (here []))))
```

```
                                   (G 0))
G (suc (suc n)) = New (Par (chan sp10 sp01 :: L [])
                                   (Send (R L []) (name (here [])) (pair (R []) (pure [] (suc (suc n))) (name (here []))))
                                   (G (suc n)))

P₄ : ℕ → Process (dual-of t₄ # _ :: [])
P₄ zero      = New (
                Par (chan sp01 sp10 :: L [])
                    (Send (R L []) (name (here [])) (pair (R []) (pure [] zero) (name (here []))))
                    (G 0)
                )
P₄ (suc n) = New (
                Par (chan sp10 sp01 :: L [])
                    (Send (R L []) (name (here [])) (pair (R []) (pure [] (suc n)) (name (here []))))
                    (G (suc n))
                )
```