Dependent Session Types via Intuitionistic Linear Type Theory

Bernardo Toninho
Carnegie Mellon University &
Universidade Nova de Lisboa
Pittsburgh, PA, USA
btoninho@cs.cmu.edu

Luís Caires
Universidade Nova de Lisboa
Lisbon, Portugal
luis.caires@di.fct.unl.pt

Frank Pfenning
Carnegie Mellon University
Pittsburgh, PA, USA
fp@cs.cmu.edu

Abstract

We develop an interpretation of linear type theory as dependent session types for a term passing extension of the π-calculus. The type system allows us to express rich constraints on sessions, such as interface contracts and proof-carrying certification, which go beyond existing session type systems, and are here justified on purely logical grounds. We can further refine our interpretation using proof irrelevance to eliminate communication overhead for proofs between trusted parties. Our technical results include type preservation and global progress, which in our setting naturally imply compliance to all properties declared in interface contracts expressed by dependent types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic

General Terms Languages, Theory

Keywords type theory, dependent types, session types, π-calculus

1. Introduction

We introduce a theory of dependent session types for distributed processes, based on an interpretation of pure linear type theory for a term passing extension of the π-calculus.

The π-calculus is a foundational model for interacting concurrent processes, building on the key ideas of naming, and name mobility. Name mobility overcame essential limitations of previous models, which were expressive enough to capture value passing concurrent computation, but not dynamic allocation and reference passing, as needed to model, e.g., ML-like programming languages and higher-order processes [26, 30]. As for the λ-calculus, the π-calculus was originally presented as an untyped language. This has opened the opportunity for intensive research on various type disciplines, some based on notions of linearity and sharing, inspired by concepts originating in linear logic [24]. More recently, session types have been introduced as a general typing discipline for name passing processes that structure interactions around the notion of sessions [20, 22].

PDPP'11, July 20–22, 2011, Odense, Denmark.

A session connects, via a private communication channel, exactly two subsystems which interact on it in perfect harmony. Interactions within a session always match precisely: when one side sends, the other receives; when one side offers a selection, the other chooses; when one side terminates, the other quits as well. Such discipline is enforced even when session channels are passed along in communications. New sessions may be dynamically created by calling on capabilities of persistent shared servers. Various forms of session types have proven useful to model realistic concurrent interactions in scenarios ranging from service-oriented computing [11] to operating system kernels [15].

In prior work [10], we have discovered a remarkable correspondence between session types and (intuitionistic) linear logic, which offers the first purely logical account of all the key features (both linear and shared) of session types. In this paper, we extend our basic interpretation to cover processes that communicate data values of an underlying functional language, not just pure sessions, and generalize it by introducing dependent types.

Our framework yields a powerful theory of dependent session types in which types may be used to specify not only the dynamics of protocols, but also properties of data received and sent in communications in the style of interface contracts. For generality, we assume data to be defined by terms of some dependent type theory, such as LF [19]. This way, functional terms may be used to represent not only basic data (such as integers, strings, structures, and higher-order functions) but also, quite importantly, proofs of data properties. Such proof terms may also be exchanged in communications, thus modeling a form of proof-carrying certification (cf. [27]), clearly useful for distributed computing. Our development is based on a purely logical foundation, via an interpretation of a standard sequent calculus proof system for linear logic [4], where base types are drawn from an underlying functional type theory [12].

All types in the logical structure are interpreted as some kind of session behavior. Following [10], multiplicative types $A \multimap B$ and $A \otimes B$, correspond to input and output session types $A?.B$, the type of sessions that receive a session of type A and then behaves as B, and $A!.B$, the type of sessions that send a session of type A and then behaves as B, respectively. The exponential type $!A$ is used to type shared channels, associated with replicated servers. As we will see, a session channel of base type $\$\tau$ just carries a basic value N of the appropriate functional type τ. A dependent type $\forall x{:}\tau.B$ types a session process that inputs a value N of type τ, and then behaves as $B\{N/x\}$. Compatibly, a type $\exists x{:}\tau.B$ types a session process that outputs a value N of type τ, and then behaves as $B\{N/x\}$. As an example, consider the process:

$$\mathsf{Up}(x) \triangleq x(n).x\langle n+1\rangle.\mathbf{0}$$

In a classical session type system, this process is given type x : ?int.!int.end, which in our basic linear session type system is rendered x : \$int $\multimap$ (\$int $\otimes$ $\mathbf{1}$). Using dependent types we can provide a much more informative interface contract, such as (among many others):

$$\mathsf{UpInterface}(x) \triangleq x : \forall n{:}\mathsf{int}.\forall p{:}(n > 0).\exists y{:}\mathsf{int}.\exists q{:}(y > 0).\mathbf{0}$$

This type specifies that if the process receives a positive amount (on session x), it will send back a positive amount as well. A sample process inhabiting type $\mathsf{UpInterface}(x)$ is

$$\mathsf{UpCert}(x) \triangleq x(n).x(p).x\langle n+1\rangle.x\langle \mathsf{incp}(n,p)\rangle.\mathbf{0}$$

Here, we have used $\mathsf{incp}(n, p)$ to denote a proof term of type $(n + 1 > 0)$, computed by some function

$$\mathsf{incp} : \Pi m{:}\mathsf{int}.(m > 0) \to (m + 1 > 0)$$

given n and p. Clearly, process $\mathsf{UpCert}(x)$ mimics $\mathsf{Up}(x)$ defined above, but also explicitly receives and sends proof certificates for the interface properties, thus witnessing the validity, at the appropriate steps, of all properties expressed by dependent types. For example, $\mathsf{UpCert}(x)$, after outputting m, also issues a proof of $(m > 0)$.

Explicitly manipulating proof certificates may be necessary in a distributed setting, but may also turn out redundant in other scenarios. To address this potential issue, again building on purely logical foundations, we explore proof irrelevance [29]. Proof irrelevance allows us to safely mark parts of a type specification that must be respected at runtime, but need not to be explicitly witnessed in the typed process. Irrelevant components A in a type are marked by a bracketing operator $[A]$. So, instead of type $\mathsf{UpInterface}(x)$ for $\mathsf{UpCert}(x)$, we may instead pick type

$$\mathsf{UpInterfaceP}(x) \triangleq x : \forall n{:}\mathsf{int}.\forall p{:}[n > 0].\exists y{:}\mathsf{int}.\exists q{:}[y > 0].\mathbf{0}$$

Then, by applying to the process $\mathsf{UpCert}(x)$ a type-directed erasure map based on $\mathsf{UpInterfaceP}(x)$, we may prune the behavior associated with irrelevant components of the process type. We then get back to the process

$$\mathsf{Up}(x) \triangleq x(n).x\langle n+1\rangle.\mathbf{0}$$

which can still be shown to conform to the rich interface type $\mathsf{UpInterfaceP}(x)$, in a precise sense, since we know the process passed type-checking with the extra information.

Our technical results show that our logical type system enjoys type preservation under reduction in a rather strong sense, and (global) progress, meaning that well typed processes never get stuck. The standard result of type preservation naturally holds in our system (Theorem 3.3). A stronger result, relating reduction in the process world and cut reduction/conversion steps in the sequent calculus world also holds, but is out of the scope of this particular presentation. The progress property (Theorem 3.5), in our setting, implies not only that all communications prescribed by types will succeed, but also that all "assertions" captured by dependent types hold at the appropriate protocol steps.

The presentation is structured as follows: In Section 2 we discuss our interpretation of linear logic as session types, beginning with a session composition principle that is embodied by a sequent calculus cut. We interpret each of the propositions of intuitionistic linear logic as session behaviors, beginning with the multiplicative fragment, followed by atomic propositions, additives and exponentials and, finally, quantifiers, which correspond to input and output of *proof terms*. Section 3 presents the results of type preservation and progress for our type system. Section 4 describes the usage of proof irrelevance as a form of type-directed runtime optimization of processes and Section 5 concludes.

2. Linear Logic as Session Types

In this section, we present our correspondence of quantified linear logic propositions as session types for a term passing π-calculus by interpreting each linear logic proposition as a type describing the session behavior of a particular channel (a summary of the process calculus definition is given in Section 2.9). The interpretation extends the one given in [10] with a functional layer, based on some dependent type theory, giving meaning to base types, and also crucially, with universal and existential dependent type constructors.

We begin by first defining our typing judgment. We start off with a single typing context Δ which is used according to a linear discipline (it is not subject to weakening or contraction). Later in the paper we add new context regions as necessary to account for the full generality of our system. Our type system assigns types to channels. The context Δ records assignments of the form $x : A$, denoting that a process typed under such an assumption expects to be placed in an environment providing the behavior A along channel x. Our typing judgment is: $\Delta \Rightarrow P :: z : A$, meaning that process P implements, on channel z, the session behavior described by A provided it is composed with a process environment that implements the behaviors specified by Δ (linearity imposes that *all* behaviors specified in Δ are completely used by P). We tacitly assume that all channels declared in Δ and the channel z are distinct. We can apply renaming as necessary to satisfy this condition. We always consider processes modulo structural congruence, therefore typing is closed under structural congruence by definition.

In existing presentations of session types [20] a notion of type duality is commonly present, in which the behavior of the inhabitants of a type is in some sense symmetric to the behavior of the inhabitants of its dual (e.g. the output session is dual to the input session, the choice session is dual to the branch session). In our setting a notion of behavioral duality also arises naturally from the additive and multiplicative nature of linear logic propositions. Multiplicative conjunction $\otimes$ and implication $\multimap$ are dual in the sense that using a session of one type is equivalent to implementing a session of the other. The same applies to additive conjunction and disjunction.

2.1 Cut as composition

A fundamental aspect of process calculi is parallel composition. Parallel composition allows for a process to rely on the functionality of another to implement its own. In our typed setting, this means that given a process P that implements behavior A along some channel x, that is, $\Delta \Rightarrow P :: x : A$, we can take a process Q that uses the behavior of type A (and maybe more) to implement the behavior C on z (formally $\Delta', x : A \Rightarrow Q :: z : C$) and compose the two processes so that the composition provides C along z outright. Since we follow a linear typing discipline, Q requires *all* the behavior supplied by P along x and therefore the composition must restrict the scope of x to the two processes. The cognoscenti will have already identified this reasoning principle as a sequent calculus cut, and we thus obtain the rule:

$$\frac{\Delta \Rightarrow P :: x : A \quad \Delta', x : A \Rightarrow Q :: z : C}{\Delta, \Delta' \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z : C}\ \mathsf{cut}$$

When we compose two processes as in the above rule, we do so in order for them to interact with one another. In general, both P and Q may perform some interaction with the outside environment, but the point of composing them together with a shared local name is so they communicate with each other and evolve together to some residual processes P' and Q'. All of these process reductions (interaction with the "outside world" by P, by Q, and interaction between P and Q) can be given meaning through the reduction of cuts in a proof. We thus take the correspondence of principal cut reductions and process reductions as a guiding principle in our

design, just as the correspondence between proof reductions and λ-calculus reductions are the guiding principle for the Curry-Howard isomorphism.

We now build up the system, following and extending [10]. We interpret linear logic propositions as types that characterize behaviors of processes as session-based interactions. The grammar of propositions is given by:

$$\begin{array}{rcl} A, B & ::= & \mathbf{1} \mid \$\tau \mid A \multimap B \mid A \otimes B \mid !A \\ & \mid & A \ \& \ B \mid A \oplus B \mid \forall x{:}\tau.B \mid \exists x{:}\tau.B \end{array}$$

2.2 Linear implication

The usual way of reading $A \multimap B$ in linear logic is that, given an A, we consume it and produce a B. Alternatively, we can think of $A \multimap B$ as *receiving* something of type A and producing something of type B. We therefore type a channel z with $A \multimap B$ as:

$$\frac{\Delta, x : A \Rightarrow P :: z{:}B}{\Delta \Rightarrow z(x).P :: z{:}A \multimap B} \ \multimap\mathsf{R}$$

Given a process that performs an input on z, binding it to x and continuing as P, we can type z with $A \multimap B$ if, under the assumption that x provides a behavior of type A, P will use that behavior to provide B along z. We have defined what it means to type a channel with $A \multimap B$, so we must now define what it means to use such a channel:

$$\frac{\Delta \Rightarrow P :: y : A \quad \Delta', x{:}B \Rightarrow Q :: z{:}C}{\Delta, \Delta', x{:}A \multimap B \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(P \mid Q) :: z{:}C} \ \multimap\mathsf{L}$$

We use a channel of type $A \multimap B$ to produce behavior C along z by first outputting a fresh name y. Since the contract of $x : A \multimap B$ dictates that x expects to receive a session that is to be used as A, we must ensure that such is indeed the case, which we do by having P provide A along y. Having given x a channel of type A, it will now provide behavior of type B, which can be used by Q to provide C along z. We can see that this interpretation is reasonable by composing an instance of $\multimap\mathsf{R}$ with an instance of $\multimap\mathsf{L}$ and appealing to our guiding principle of corresponding process reductions with cut reductions (we omit the full typing contexts for brevity):

$$\dfrac{\dfrac{y{:}A \Rightarrow P :: x : B}{\Rightarrow x(y).P :: x{:}A \multimap B} \quad \dfrac{\Rightarrow Q_1 :: y{:}A \quad x{:}B \Rightarrow Q_2 :: z{:}C}{x{:}A \multimap B \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(Q_1 \mid Q_2) :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(x(y).P \mid (\boldsymbol{\nu} y)x\langle y\rangle.(Q_1 \mid Q_2)) :: z{:}C}$$

$$\longrightarrow \quad \dfrac{\dfrac{\Rightarrow Q_1 :: y{:}A \quad y{:}A \Rightarrow P :: x{:}B}{\Rightarrow (\boldsymbol{\nu} y)(Q_1 \mid P) :: x{:}B} \quad x : B \Rightarrow Q_2 :: z{:}C}{\Rightarrow (\boldsymbol{\nu} x)((\boldsymbol{\nu} y)(Q_1 \mid P) \mid Q_2) :: z{:}C}$$

We can isolate the process reduction induced by this cut reduction

$$(\boldsymbol{\nu} x)(x(y).P \mid (\boldsymbol{\nu} y)x\langle y\rangle.(Q_1 \mid Q_2)) \longrightarrow (\boldsymbol{\nu} x)((\boldsymbol{\nu} y)(Q_1 \mid P) \mid Q_2)$$

and observe that, modulo structural congruence, it is the expected interaction between an input process $x(y).P$ and output process $x\langle y\rangle.(Q_1 \mid Q_2)$ along a private channel x.

2.2.1 A simple example

Consider we want to describe a bank service in our system. With what we have presented so far, we can specify what is, for the moment, the protocol of a very simple bank process that receives a string encoding a user's identification and an amount that is to be deposited and just terminates:

$$\mathsf{TBank} \triangleq \$\mathsf{string} \multimap (\$\mathsf{nat} \multimap \mathbf{1})$$

We have not yet introduced base types (such as $\$\mathsf{string}$), but we will get into that shortly. The multiplicative unit $\mathbf{1}$, as we show in the following section, denotes the terminated session. An example of a process providing a session of this type on channel x is:

$$x(s).x(n).\mathbf{0} :: x : \mathsf{TBank}$$

This is not *yet* a particularly interesting example. However, as we interpret more linear logic connectives, we can gradually refine our bank specification to describe richer and more interesting features.

2.3 Multiplicative unit

The multiplicative unit of intuitionistic linear logic, written $\mathbf{1}$, is a proposition that is proved using no resources. Dually, using the unit just consumes it, providing no resources. In a process calculus setting, we interpret $\mathbf{1}$ as the terminated session:

$$\frac{}{\cdot \Rightarrow \mathbf{0} :: z : \mathbf{1}} \ \mathbf{1}\mathsf{R} \qquad \frac{\Delta \Rightarrow P :: z : C}{\Delta, x : \mathbf{1} \Rightarrow P :: z : C} \ \mathbf{1}\mathsf{L}$$

We provide a session of type $\mathbf{1}$ with the terminated process (it uses no further ambient resources) and use it (if such is even the appropriate term) by simply erasing. This is one of the two cases where no process reduction takes place in composition, since the inactive process and the scope restriction are erased through *structural congruence*, not through reduction:

$$\dfrac{\Rightarrow 0 :: x : \mathbf{1} \quad \dfrac{\Rightarrow P :: z : C}{x : \mathbf{1} \Rightarrow P :: z : C}}{\Rightarrow (\nu x)(\mathbf{0} \mid P) :: z : C} \qquad \equiv \quad \Rightarrow P :: z : C$$

Note that in terms of behavior duality, $\mathbf{1}$ is self-dual.

2.4 Multiplicative conjunction

Multiplicative conjunction, written $A \otimes B$, means that we must be able to divide our resources (in our interpretation, the sessions available for interaction in the context) in such a way that we can produce both an A and a B. In fact, the rules for $\otimes$ exhibit a deep symmetry with those for linear implication ($\otimes$ is behaviorally dual to $\multimap$ in the manner explained in the introduction of Section 2). We exploit this symmetry and interpret $\otimes$ on the right as output and as input on the left:

$$\frac{\Delta \Rightarrow P :: y : A \quad \Delta' \Rightarrow Q :: z : B}{\Delta, \Delta' \Rightarrow (\boldsymbol{\nu} y)z\langle y\rangle.(P \mid Q) :: z : A \otimes B} \ \otimes\mathsf{R}$$

Since we need to able to provide both session behaviors A and B, we output a fresh channel y, through which the process P provides a session of type A. Since we are already communicating along z, we use it to provide a session of type B, which is realized by process Q. We use a session of type $A \otimes B$ as follows:

$$\frac{\Delta, y : A, x : B \Rightarrow P :: z : C}{\Delta, x : A \otimes B \Rightarrow x(y).P :: z : C} \ \otimes\mathsf{L}$$

We input along x, because the contract of $x : A \otimes B$ enforces that an output of a channel which can be used as a session of type A will take place on x, we bind that channel to y, and we can then safely use x as providing type B to provide C along z. The reduction that supports this interpretation is:

$$\dfrac{\dfrac{\Rightarrow P_1 :: y{:}A \quad \Rightarrow P_2 :: x{:}B}{\Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(P_1 \mid P_2) :: x{:}A \otimes B} \quad \dfrac{y{:}A, x{:}B \Rightarrow P :: z{:}C}{x{:}A \otimes B \Rightarrow x(y).Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)((\boldsymbol{\nu} y)x\langle y\rangle.(P_1 \mid P_2) \mid x(y).Q) :: z{:}C}$$

$$\longrightarrow \quad \dfrac{\Rightarrow P_2 :: x{:}B \quad \dfrac{\Rightarrow P_1 :: y{:}A \quad y{:}A, x{:}B \Rightarrow Q :: z{:}C}{x{:}B \Rightarrow (\boldsymbol{\nu} y)(P_1 \mid Q) :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(P_2 \mid (\boldsymbol{\nu} y)(P_1 \mid Q)) :: z{:}C}$$

Again, modulo structural congruence, this is exactly the appropriate process reduction, communicating along the private channel x.

2.4.1 A slightly less simple example

The example of 2.2.1 consists of a bank specification that only allows a client to send its user identification, an amount to be deposited and then terminate. Now that we have available the $\otimes$ type, we can slightly enrich our bank to send back to the client a receipt of the deposited amount:

$$\mathsf{TBank} \triangleq \$\mathsf{string} \multimap (\$\mathsf{nat} \multimap (\$\mathsf{nat} \otimes \mathbf{1}))$$

For which we can produce the process:

$$z(s).z(a).(\nu r)z\langle r\rangle.(P_{\mathsf{receipt}} \mid \mathbf{0}) :: z : \mathsf{TBank}$$

where P_{receipt} is a process that will return an appropriate receipt back to the client. In order to give a precise definition of P_{receipt} we need to develop a way of mentioning basic values such as numbers, which we do in the following section.

Note, however, that this is still a rather simplistic bank process in that it only offers deposit operations (which would not leave its clients very happy), and only runs once. Moreover, this specification only really guarantees that the bank will send back a number. Nothing ensures that it really corresponds to the same value that the client wanted to deposit. In the following sections we develop our system to adress each of these issues, ultimately building up to the a dependent linear type theory of sessions.

2.5 Base types and the identity rule

In the previous section we have shown how to interpret linear implication and conjunction as the types of input and output sessions, respectively. Before proceeding to the remaining linear logic connectives, we will assign meaning to base types and interpret the identity axiom of linear logic. As we have hinted at in the previous example, these turn out to be essential for our development.

A base type $\$\tau$ denotes a proposition that can only be ultimately proved from an ambient assumption of that particular type because it cannot be decomposed further. In this sense, $\$\tau$ is an *atom*. Moreover, linear logic only allows us to prove $\$\tau$ if it is our only remaining resource. In previous work [10], since the focus was on interpreting the composite connectives as pure process behavior, no interpretation was given for atomic types. Here, atomic types connect us to another language layer.

Commonly, we want processes to exchange data, such as numbers and strings (indeed, most work on session types takes this for granted and assumes that processes exchange channels and data values [7, 8, 18, 22]). In our approach processes communicate not just names, but also terms of a *functional* language that assigns meaning to the base types of the full calculus and, as we show in Section 2.8, produces the witnesses for universally and existentially quantified types.

Note that while these terms populate base types, the types need not actually be atomic in the term language. Any extra type structure only has meaning in the term language, while from the perspective of the process calculus they are opaque types with no further decomposable structure. Letters M, N range over the terms of this language and we rely on a separate judgment for well-formedness of such terms, written $\Psi \vdash M : \tau$. Ψ is a context region that is reserved for the term language (we may trivially add the context Ψ to all the sequents in the rules we have seen so far, since these do not affect Ψ). Note that τ itself has meaning in the functional language, while in the process calculus all such types are internalized as $\$\tau$.

We refrain from fully specifying the term language to maintain full generality. We instead assume that the term language is defined by some intuitionistic system of natural deduction with the usual properties of substitution and weakening (we could relax the requirement of weakening by considering a typed linear lambda calculus as the term language such as [12], but we refrain from doing so for simplicity of presentation).

We only require two additional rules to fully account for base types and the corresponding terms of the functional language:

$$\frac{\Psi \vdash M : \tau}{\Psi; \cdot \Rightarrow [z \leftarrow M] :: z : \$\tau}\ \$\mathsf{R}$$

The $\$\mathsf{R}$ rule allows us to use terms from the functional language to give meaning to names at base type. The process construct $[z \leftarrow M]$ locates functional term M at name z (we will introduce the operational semantics for this construct shortly). The final missing piece is a rule that takes names of base type from the linear context and places them in the appropriate Ψ context:

$$\frac{\Psi, x : \tau; \Delta \Rightarrow P :: z : C}{\Psi; \Delta, x : \$\tau \Rightarrow P :: z : C}\ \$\mathsf{L}$$

This rule realizes our design to give meaning to base types in the external functional language: given a channel that provides complex session behavior, we successively play out the session down to its basic constituents (which are the types of the functional term language, the terminated session $\mathbf{1}$ or, as we detail later, persistent sessions), at which point, if we are in the presence of a base type, we move it to the context Ψ where it can be further interpreted as needed.

We can now determine what the behavior of the construct in the $\$\mathsf{R}$ rule should be:

$$\frac{\dfrac{\vdash M : \tau}{\Rightarrow [x \leftarrow M] :: x : \$\tau} \quad \dfrac{x : \tau; \cdot \Rightarrow P :: z : C}{x : \$\tau \Rightarrow P :: z : C}}{\Rightarrow (\boldsymbol{\nu} x)([x \leftarrow M] \mid P) :: z : C}$$

$$\longrightarrow \Rightarrow P\{M/x\} :: z : C$$

where $P\{M/x\}$ is the substitution of term M for variable x in P. The construct $[x \leftarrow M]$ is reminiscent of the applied π-calculus notion of active substitution [1]. In the applied π-calculus, there is no reduction step like the one above, and the substitution is instead silently performed by a structural congruence principle. Although we might have alternatively interpreted this cut-elimination step by a structural congruence (as we have done for multiplicative unit), we prefer not to do so, without any loss of generality, to maintain a crisper correspondence with the dynamics suggested by the proof theory.

2.5.1 Identity as renaming

We have stated that hypotheses denote the existence of ambient names providing certain behaviors. On the logical side, initial sequents $\Psi; x : A \Rightarrow P :: z : A$ allow us to use an assumption directly to prove the conclusion, a rule absent in [10]. We know that x stands for a name or a term of type A, whatever it may be, and we want to make use of x to provide that same A as z. We thus want to *equate* x and z as the *same*, and that is precisely the behavior that process P must implement. For this we introduce a new process construction, $[x \leftrightarrow z]$, meaning that both names are interchangeable, obtaining the rule:

$$\frac{}{\Psi; x : A \Rightarrow [x \leftrightarrow z] :: z : A}\ \mathsf{id}$$

The proof reductions that we obtain in cut elimination can inform us of what the reductions should be:

$$\frac{y : A \Rightarrow [y \leftrightarrow x] :: x : A \quad x : A \Rightarrow P :: z : C}{y : A \Rightarrow (\boldsymbol{\nu} x)([y \leftrightarrow x] \mid P) :: z : C}$$

$$\longrightarrow y : A \Rightarrow P\{y/x\} :: z : C$$

$$\frac{\Rightarrow P :: x : A \quad x : A \Rightarrow [x \leftrightarrow z] :: z : A}{\Rightarrow (\boldsymbol{\nu} x)(P \mid [x \leftrightarrow z]) :: z : A}$$

$$\longrightarrow \quad \Rightarrow P\{z/x\} :: z : A$$

And so interchangeable names will, operationally, be substituted for each other. We are justified in renaming one to the other in a type-safe way. It is possible to replace this construct at any composite type by a process that acts as an intermediary between the ambient session and the provided one, simply acting as a copycat process, until we reach a base type, at which point the two names are equated to refer to the same functional term. This is the computational content of the meta-theoretic proof of admissibility of the identity rule (or *initial* rule) in a sequent calculus.

The two rules above define the reduction of the renaming construct with the proviso that y and z do not occur in P, respectively. In general, we impose the formation restriction that one of the names appearing in the renaming construct must be bound, while the other one must not occur within the remaining scope of the renaming construct, which is enforced by our typing discipline. By adding a structural congruence, $[y \leftrightarrow x] \equiv [x \leftrightarrow y]$, we can summarize the two rules as one:

$$(\boldsymbol{\nu} x)([y \leftrightarrow x] \mid P) \longrightarrow P\{y/x\}$$

2.6 Additive conjunction and disjunction

We now turn our attention to additive conjunction, written $A \,\&\, B$. Additive conjunction represents alternative availability of resources (we are prepared to provide sessions A and B, but can only provide one of them), where the choice of resource A or B is made by the client of $A \,\&\, B$. We thus type a channel with $A \,\&\, B$ if it *offers* a choice between the two behaviors A and B:

$$\frac{\Psi; \Delta \Rightarrow P :: z : A \quad \Psi; \Delta \Rightarrow Q :: z : B}{\Psi; \Delta \Rightarrow z.\mathsf{case}(P, Q) :: z : A \,\&\, B}\ \&\mathsf{R}$$

The process above branches to provide either A or B. If A is selected, the process P provides the necessary session behavior along z, otherwise, process Q provides the session behavior B along z. We can use a channel of type $A \,\&\, B$ by triggering either one of the possible choices:

$$\frac{\Psi; \Delta, x : A \Rightarrow P :: z : C}{\Psi; \Delta, x : A \,\&\, B \Rightarrow x.\mathsf{inl}; P :: z : C}\ \&\mathsf{L}_1$$

$$\frac{\Psi; \Delta, x : B \Rightarrow P :: z : C}{\Psi; \Delta, x : A \,\&\, B \Rightarrow x.\mathsf{inr}; P :: z : C}\ \&\mathsf{L}_2$$

This form of minimal labeled choice is comparable to the n-ary branching constructs of standard session-oriented π-calculi [22]. The behavioral dual of binary branching is binary choice, which corresponds to additive disjunction:

$$\frac{\Psi; \Delta \Rightarrow P :: z : A}{\Psi; \Delta \Rightarrow z.\mathsf{inl}; P :: z : A \oplus B}\ \oplus\mathsf{R}_1$$

$$\frac{\Psi; \Delta \Rightarrow P :: z : B}{\Psi; \Delta \Rightarrow z.\mathsf{inr}; P :: z : A \oplus B}\ \oplus\mathsf{R}_2$$

This means that in order to use a session of type $A \oplus B$ to offer a session behavior of type C, we must be able to offer C for both possibilities of the choice:

$$\frac{\Psi; \Delta, x : A \Rightarrow P :: z : C \quad \Psi; \Delta, x : B \Rightarrow Q :: z : C}{\Psi; \Delta, x : A \oplus B \Rightarrow x.\mathsf{case}(P, Q) :: z : C}\ \oplus\mathsf{L}$$

The reduction we obtain through composition is:

$$\frac{\dfrac{\Rightarrow P_1 :: x{:}A \quad \Rightarrow P_2 :: x{:}B}{\Rightarrow x.\mathsf{case}(P_1, P_2) :: x{:}A \,\&\, B} \qquad \dfrac{x{:}A \Rightarrow Q :: z{:}C}{x{:}A \,\&\, B \Rightarrow x.\mathsf{inl}; Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(x.\mathsf{case}(P_1, P_2) \mid x.\mathsf{inl}; Q) :: z{:}C}$$

$$\longrightarrow \quad \frac{\Rightarrow P_1 :: x{:}A \quad x{:}A \Rightarrow Q :: z{:}C}{\Rightarrow (\boldsymbol{\nu} x)(P_1 \mid Q) :: z{:}C}$$

and symmetrically:

$$\frac{\dfrac{\Rightarrow P_1 :: x{:}A \quad \Rightarrow P_2 :: x{:}B}{\Rightarrow x.\mathsf{case}(P_1, P_2) :: x{:}A \,\&\, B} \qquad \dfrac{x{:}B \Rightarrow Q :: z : C}{x{:}A \,\&\, B \Rightarrow x.\mathsf{inr}; Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(x.\mathsf{case}(P_1, P_2) \mid x.\mathsf{inr}; Q) :: z{:}C}$$

$$\longrightarrow \quad \frac{\Rightarrow P_1 :: x{:}B \quad x{:}B \Rightarrow Q :: z{:}C}{\Rightarrow (\boldsymbol{\nu} x)(P_2 \mid Q) :: z{:}C}$$

2.6.1 A slightly less simple example... with choice

We refine our previous bank specification to account for the fact that a bank offers several possible operations to its clients. In particular, we consider the deposit operation of Section 2.4.1 and consulting the account balance:

$$\mathsf{TBank} \triangleq \$\mathsf{string} \multimap ((\$\mathsf{nat} \multimap (\$\mathsf{nat} \otimes \mathbf{1})) \,\&\, (\$\mathsf{nat} \otimes \mathbf{1}))$$

We abstract the details of performing the deposit operation with a function $\mathsf{dep} : \mathsf{string} \to \mathsf{nat} \to \mathsf{nat}$ that takes the user identification and the deposit amount and returns the receipt, and the details of obtaining the balance of an account with a function $\mathsf{bal} : \mathsf{string} \to \mathsf{nat}$ that takes the user identification and returns the balance of the account:

$$z(s).z.\mathsf{case}(z(a).(\nu r)z\langle r\rangle.([r \leftarrow \mathsf{dep}(s, a)] \mid \mathbf{0}),\\ (\nu b)z\langle b\rangle.([b \leftarrow \mathsf{bal}(s)] \mid \mathbf{0}) :: z : \mathsf{TBank}$$

2.7 Replication and exponential

We now develop the technical apparatus to provide an interpretation of the linear logic exponential $!A$. Proof-theoretically, the exponential enables a form of controlled weakening and contraction. More precisely, a proposition $!A$ provides an arbitrary number of copies of A (possibly 0). This means that to prove $!A$, we cannot use *any* linear resource, otherwise we would not be able to use A an arbitrary number of times. To cleanly account for the ability to weaken and contract certain resources, we split the context in an unrestricted zone that is subject to weakening and contraction, which we call Γ, and the linear zone (not subject to weakening or contraction), which we still denote as Δ (this form of context splitting is consistent with Barber and Plotkin's DILL [4]). Variables declared in Γ are called *unrestricted* and are denoted by (u, v, w). As before with the context Ψ, we simply add Γ to all sequents in the rules we have presented so far, since they do not use or change Γ in any way.

We can now assign the type $!A$ to a channel z as follows:

$$\frac{\Psi; \Gamma; \cdot \Rightarrow P :: y : A}{\Psi; \Gamma; \cdot \Rightarrow !z(y).P :: z : !A}\ !\mathsf{R}$$

We represent the *persistent* (or unrestricted) nature of the exponential by using an input-guarded process replication construct. The above process expects an input along z (call it y) to trigger the replication. The received name y will be the one through which P provides the session behavior of type A. Since the input is replicated (and P does not depend on any *linear* sessions), the process is able to provide an arbitrary number of copies of the session behavior A. Note that while we do require the linear context to be

empty, we can use any ambient persistent session channel (called *standard* channels in [17]) in Γ to implement a session of type $!A$.

Using a (linear!) channel x of type $!A$ conceptually requires two steps. The first is to unlock the ability for this channel to provide session A multiple times. This is accomplished simply by renaming, taking care to make sure that the new channel $u : A$ is persistent and therefore declared in Γ.

$$\frac{\Psi; \Gamma, u : A; \Delta \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : !A \Rightarrow P\{x/u\} :: z : C}\ !\mathsf{L}$$

The second step is to actually create a fresh channel $y : A$ while retaining the capability to create more in the future, encoded by keeping $u : A$ in the context.

$$\frac{\Psi; \Gamma, u : A; \Delta, y : A \Rightarrow P :: z : C}{\Psi; \Gamma, u : A; \Delta \Rightarrow (\boldsymbol{\nu} y) u\langle y\rangle.P :: z : C}\ \mathsf{copy}$$

This *copy* rule is characteristic of sequent calculi implementing DILL. It is interesting that !L merely renames, while copy outputs a new bound name, being the computationally significant operation.

To follow our program of identifying process reductions with principal cut reductions, we must first observe that our previous composition rule cut cannot properly account for ambient unrestricted assumptions and thus does not completely explain typed composition in its full generality. In fact, if we simply compose the instances of !R and !L using cut:

$$\frac{\dfrac{\Psi; \Gamma; \cdot \Rightarrow P :: y{:}A}{\Psi; \Gamma; \cdot \Rightarrow !x(y).P :: z{:}!A} \quad \dfrac{\Psi; \Gamma, u{:}A; \Delta \Rightarrow Q :: z{:}C}{\Psi; \Gamma; \Delta, x{:}!A \Rightarrow Q\{x/u\} :: z{:}C}}{\Psi; \Gamma; \Delta \Rightarrow (\boldsymbol{\nu} x)(!x(y).P \mid Q\{x/u\}) :: z{:}C}$$

not only can we not produce a process reduction (which is expected due to the "silent" nature of !L), but we also are unable to produce a proof reduction, since up to this point we have not defined a persistent version of cut. We can fix this by considering a composition rule for unrestricted sessions:

$$\frac{\Psi; \Gamma; \cdot \Rightarrow P :: x : A \quad \Psi; \Gamma, u : A; \Delta \Rightarrow Q :: z : C}{\Psi; \Gamma; \Delta \Rightarrow (\boldsymbol{\nu} u)(!u(x).P \mid Q) :: z : C}\ \mathsf{cut}^!$$

Given a process P that provides a session A along x without using any ambient linear sessions, and a process Q that implements session behavior C along z by (potentially) using the unrestricted ambient session $u : A$ (as well as linear ambient sessions Δ), we may compose Q with P if we prepend a replicated input along u to P, so it may now provide the necessary multiple copies of the session behavior A to produce a process that provides C along z outright. We can now exhibit our correspondence on the copy rule, where the process reduction is matched with a proof reduction obtained by the elimination of a persistent cut:

$$\frac{\Rightarrow P :: x{:}A \quad \dfrac{u{:}A; x{:}A \Rightarrow Q :: z{:}C}{u{:}A; \Rightarrow (\boldsymbol{\nu} x) u\langle x\rangle.Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} u)(!u(x).P \mid (\boldsymbol{\nu} x) u\langle x\rangle.Q) :: z{:}C} \quad \longrightarrow$$

$$\frac{\Rightarrow P :: x{:}A \quad \dfrac{u{:}A \Rightarrow P :: x{:}A \quad u{:}A; x{:}A \Rightarrow Q :: z{:}C}{u{:}A \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} u)(!u(x).P \mid (\boldsymbol{\nu} x)(P \mid Q)) :: z{:}C}$$

If we now revisit our previous composition of !R and !L, we can observe that the process composition is structurally equivalent to persistent composition (which we know to exhibit the appropriate process reduction when the persistent session u is actually used). Similarly to what happens with $\mathbf{1}$, this is also one of the situations where we witness a proof reduction (of a cut to a persistent cut) that is matched by structural congruence in the process calculus. Note that the proof reductions of the persistent cut are again matched by process reductions (as we have shown above).

This form of composition of unrestricted resources introduces a proof conversion in which the unrestricted resource is "garbage collected" if never used. We can interpret this conversion as extending the standard structural congruence $\equiv$ between processes with the following rule (we will refer to this extended congruence as $\equiv_S$):

$$(\boldsymbol{\nu} x)(!x(y).P \mid Q) \equiv_S Q \text{ if } x \notin \mathit{fn}(Q)$$

While not essential to our development, $\equiv_S$ allows us to provide a more concise statement for some of the theorems of Section 3.

2.7.1 A bank with a persistent service

Having properly defined persistent sessions through linear logic exponentials, we can now have a bank service that persists through multiple sessions, instead of just being available for one usage:

$$\mathsf{TBank} \triangleq !(\$\mathsf{string} \multimap ((\$\mathsf{nat} \multimap (\$\mathsf{nat} \otimes \mathbf{1})) \,\&\, (\$\mathsf{nat} \otimes \mathbf{1})))$$

We modify the bank process to be

$$!z(y).y(s).y.\mathsf{case}(y(a).(\nu r) y\langle r\rangle.([r \leftarrow \mathsf{dep}(s, a)] \mid \mathbf{0}),\\ (\nu b) y\langle b\rangle.([b \leftarrow \mathsf{bal}(s)] \mid \mathbf{0})) :: z : \mathsf{TBank}$$

which now receives a session channel (bound to y) and spawns a replica that provides the behavior $\$\mathsf{string} \multimap ((\$\mathsf{nat} \multimap (\$\mathsf{nat} \otimes \mathbf{1})) \,\&\, (\$\mathsf{nat} \otimes \mathbf{1}))$ along y.

We now have what may seem to be a good specification for what a bank process should be. However, if we only consider the type TBank, we are really only describing a persistent service that will receive a string and give a choice between either receiving a number and sending one back or just sending a number. When seen under this light, it becomes less obvious that we should be happy with our specification of what a simple bank process should be. In the next section, we develop a way of refining the specification such that typing will ensure strong guarantees not just on the pure session behavior, but also on the relationships between the actual communicated data. This refinement comes from the universal and existential quantifiers of linear logic, which are interpreted as a form of dependent product and sum, respectively.

2.8 Quantification and term passing

In intuitionistic first-order linear logic we usually consider the quantifiers $\forall x.A$ and $\exists x.A$ as ranging over a single domain that is left unspecified in order to study quantification in a general setting, independent of a particular domain of discourse. We now reconsider the quantifiers as $\forall x{:}\tau.A$ and $\exists x{:}\tau.A$, and therefore focus on quantification where the domain of discourse is *typed* (in particular, with a type τ).

Let us first consider universal quantification. Logic allows us to conclude $\forall x{:}\tau.A$ if by hypothesizing the existence of some element of type τ, labeled by x, we can prove A (which may *depend* on x). In linear logic, the hypothesis $x : \tau$ is given an unrestricted character since it avoids the problematic situation where a proposition may refer to an object that may have already been consumed. Conversely, we use an assumption of $\forall x{:}\tau.A$ by providing an object of type τ, which enables us to use A with the free variable x appropriately instantiated (in type theory this means that A *depends* on a term of type τ). We thus interpret a channel of type $\forall x{:}\tau.A$ as follows:

$$\frac{\Psi, x : \tau; \Gamma; \Delta \Rightarrow P :: z : A}{\Psi; \Gamma; \Delta \Rightarrow z(x).P :: z : \forall x : \tau.A}\ \forall\mathsf{R}$$

Similarly to how in type theory the universal quantifier corresponds to implication, we type the name z with $\forall x{:}\tau.A$ if after performing an input of a *term* of type τ, we can type z with A in the continuation P. We now define how to use a name of type $\forall y{:}\tau.A$:

$$\frac{\Psi \vdash N : \tau \quad \Psi; \Gamma; \Delta, x : A\{N/y\} \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : \forall y : \tau.A \Rightarrow x\langle N\rangle.P :: z : C}\ \forall\mathsf{L}$$

To use an ambient channel x of this type, we must output a functional term of type τ. Upon doing so, x now offers the session A, where the free variable in A has been instantiated with the term N, which we can use in P to provide session C along z.

We choose to use functional terms as the quantifier witnesses because they allow us to refer to the values communicated by processes (which are defined by the same functional language). This allows us to express rich properties of the values communicated by processes (which we will see shortly). Furthermore, it allows us to give a clean and logically based account of processes that exchange proof objects (i.e., the functional terms) which can serve as a form of inspectable proof certificate (vis., a high-level model of proof carrying code [27]).

The reduction for the processes in $\forall\mathsf{R}$ and $\forall\mathsf{L}$ is:

$$\dfrac{\dfrac{y{:}\tau;-;-\Rightarrow P :: x{:}A}{\Rightarrow x(y).P :: x{:}\forall y{:}\tau.A} \quad \dfrac{\vdash N{:}\tau \quad x{:}A\{N/y\}\Rightarrow Q :: z{:}C}{x : \forall y{:}\tau.A \Rightarrow x\langle N\rangle.Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(x(y).P \mid x\langle N\rangle.Q) :: z{:}C}$$

$$\longrightarrow \dfrac{\Rightarrow P\{N/y\} :: x{:}A\{N/y\} \quad x{:}A\{N/y\}\Rightarrow Q :: z{:}C}{\Rightarrow (\boldsymbol{\nu} x)(P\{N/y\} \mid Q) :: z{:}C}$$

We now consider existential quantification. Logic allows us to conclude $\exists x{:}\tau.A$ if we can produce a witness of type τ and (potentially) use it to show A (in which x may be free and therefore we need to instantiate the variable x with the witness). Just as universal quantification was interpreted as term input, we interpret existential quantification as its behavioral dual, that is, as term output:

$$\dfrac{\Psi \vdash N : \tau \quad \Psi;\Gamma;\Delta \Rightarrow P :: z : A\{N/x\}}{\Psi;\Gamma;\Delta \Rightarrow z\langle N\rangle.P :: z : \exists x{:}\tau.A}\ \exists\mathsf{R}$$

The term N provides a witness of τ, which is used to instantiate x in the session type A provided by P along z. Using a channel of type $\exists y{:}\tau.A$ is defined as:

$$\dfrac{\Psi, y : \tau;\Gamma;\Delta, x : A \Rightarrow P :: z : C}{\Psi;\Gamma;\Delta, x : \exists y{:}\tau.A \Rightarrow x(y).P :: z : C}\ \exists\mathsf{L}$$

Given that the contract of $x : \exists y{:}\tau.A$ is to output a term of type τ along x and then provide behavior A (with the appropriate instantiation of the variable y), we use a session of existential type by performing an input along x, that is bound in the continuation as y, which then uses the residual behavior A to provide C along z.

The reduction of the process composition is identical to that for sessions of universal quantification type:

$$\dfrac{\dfrac{\vdash N{:}\tau \quad \Rightarrow P :: x{:}A\{N/y\}}{\Rightarrow x\langle N\rangle.P :: x{:}\exists y{:}\tau.A} \quad \dfrac{y{:}\tau; x{:}A \Rightarrow Q :: z{:}C}{x{:}\exists y{:}\tau.A \Rightarrow x(y).Q :: z{:}C}}{\Rightarrow (\boldsymbol{\nu} x)(x\langle N\rangle.P \mid x(y).Q) :: z{:}C}$$

$$\longrightarrow \dfrac{\Rightarrow P :: x{:}A\{N/y\} \quad x{:}A \Rightarrow Q\{N/y\} :: z{:}C}{\Rightarrow (\boldsymbol{\nu} x)(P \mid Q\{N/y\}) :: z{:}C}$$

We must note that as of this moment in our presentation, our system is not yet a truly dependent type theory of sessions, since we have not yet defined a way in which we can actually have occurrences of the quantified variables in the bodies of types. In logic, this is achieved by allowing atomic propositions p to depend on (typed) variables, that is, to have atomic propositions be predicates on typed objects (e.g. in $\forall x{:}\tau.p(x)$, p is a predicate on objects of type τ). In type theory, predicates correspond to indexed families of types. For instance, $\forall x{:}\tau.p(x)$ defines a type family p indexed by objects of type τ, that is, $p(N)$ is a type for any object N of type τ. We refrain from presenting further insights into the technical aspects of dependent type theories for the sake of brevity, simply noting that their expressive power gives rise to practical and useful solutions to problems that range from foundational aspects [13, 14, 25] to more practical aspects of computer science [28, 31].

In our interpretation, we assume that we can define type families in the functional term language, that is, the functional term language is a dependent type theory in the style of [19, 28]. We thus introduce the final requirement that makes our interpretation a fully dependent type theory of sessions.

2.8.1 A more sophisticated bank service

We now extend our running example of the bank process to a system with a bank and an ATM that interfaces between the bank and its clients. The ATM charges any client a small amount for any operations performed. We therefore specify such an ATM, with the additional caveat that it may only charge at most 2 dollars per operation, and it must provide a proof of such to the client. We begin with the bank specification:

$$\begin{aligned}\mathsf{TBank} \triangleq\ !(\forall s : \mathsf{string}.\$\mathsf{uid}(s) \multimap\ & \\ (\forall n : \mathsf{nat}.\$\mathsf{deposit}(s,n) \multimap (\$\mathsf{receipt}(s,n) \otimes \mathbf{1})) \ \&\ & \\ (\exists m : \mathsf{nat}.\$\mathsf{balance}(s,m) \otimes \mathbf{1}))\end{aligned}$$

By using dependent types at both the session level and at the functional term level, we can provide a refined specification in which the bank receives the user identification and then offers the deposit and balance operations: the former receives a deposit order of n dollars for the specified user s and issues a receipt that refers to s and n (all of which is ensured by typing); the latter simply issues a balance statement that refers to s and an amount m corresponding to the account balance. We use dependent functions dep with type $\Pi s : \mathsf{string}.\Pi n : \mathsf{nat}.\mathsf{deposit}(s,n) \to \mathsf{receipt}(s,n)$ and bal with type $\Pi s : \mathsf{string}.\Sigma m : \mathsf{nat}.\mathsf{balance}(s,m)$ to implement the bank process ($\pi_i(N)$ denotes the ith projection of N):

$$\begin{aligned}&!z(y).y(s).y(id).y.\mathsf{case}(y(n).y(d).(\boldsymbol{\nu} r)y\langle r\rangle. \\ &\quad ([r \leftarrow \mathsf{dep}(s,n,d)] \mid \mathbf{0}), y\langle \pi_1(\mathsf{bal}(s))\rangle.(\boldsymbol{\nu} b)y\langle b\rangle. \\ &\qquad ([b \leftarrow \pi_2(\mathsf{bal}(s))] \mid \mathbf{0})) :: z : \mathsf{TBank}\end{aligned}$$

The ATM client interface specification is (to make matters simpler, we assume the ATM only performs deposits):

$$\begin{aligned}\mathsf{TATMClient} \triangleq\ & \forall s : \mathsf{string}.\$\mathsf{uid}(s) \multimap \\ & (\forall n : \mathsf{nat}.\$\mathsf{deposit}(s,n) \multimap \exists m : \mathsf{nat}. \\ & \exists p : (\mathsf{n} - 2 \leq \mathsf{m} \leq \mathsf{n}).(\$\mathsf{receipt}(s,m) \otimes \mathbf{1}))\end{aligned}$$

The client sends its user id, a deposit instruction for some amount n, and the ATM sends back to the client the receipt for the deposited amount, along with a proof object p that guarantees that the amount charged for the deposit is within the bounds imposed by the specification. Note that we can now ensure by typing alone that any well-typed ATM will be guaranteed to *not* overcharge its clients. For the ATM process, we use a function charge of type:

$$\begin{aligned}\mathsf{charge} : \Pi s{:}\mathsf{string}.\Pi n{:}\mathsf{nat}.\mathsf{deposit}(s,n) \to \\ \Sigma m : \mathsf{nat}.\Sigma p : (n - 2 \leq m \leq n).\mathsf{deposit}(s,m)\end{aligned}$$

The charge function takes the deposit object and issues a new deposit object, providing the necessary proof objects to ensure that the amount charged for the operation is within specification bounds. An inhabitant of type $\mathsf{TATMClient}$ (assuming the bank session is available on channel x) is:

$$\begin{aligned}&z(s).z(id).z(n).z(d). \\ &\quad (\boldsymbol{\nu} y)x\langle y\rangle.y\langle s\rangle.(\boldsymbol{\nu} i)y\langle i\rangle.([i \leftarrow id] \mid \\ &\qquad y.\mathsf{inl}; y\langle \pi_1(\mathsf{charge}(s,n,d))\rangle. \\ &\qquad\quad (\boldsymbol{\nu} d')y\langle d'\rangle.([d' \leftarrow \pi_2(\pi_2(\mathsf{charge}(s,n,d)))] \mid \\ &\qquad\qquad y(r).z\langle \pi_1(\mathsf{charge}(s,n,d))\rangle. \\ &\qquad\qquad\quad z\langle \pi_1(\pi_2(\mathsf{charge}(s,n,d)))\rangle. \\ &\qquad\qquad\qquad (\boldsymbol{\nu} t)z\langle t\rangle.([t \leftrightarrow r] \mid 0))) :: z : \mathsf{TATMClient}\end{aligned}$$

Note that there are potentially several inhabitants of the type $\mathsf{TATMClient}$, due to the many possible ways in which the com-

munication on the bank session channel x and the client session channel z can be validly interleaved (e.g. the ATM might send the proof objects to the client before sending the deposit message to the bank).

2.9 Summary

We now take a step back and summarize. We have presented a type system of dependent session types for a term passing π-calculus, whose process constructors are given below:

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid P|Q \mid (\boldsymbol{\nu} y)P \\ & & \mid x\langle y\rangle.P \mid x\langle N\rangle.P \mid x(y).P \\ & & \mid !x(y).P \mid x.\mathsf{inl}; P \mid x.\mathsf{inr}; P \\ & & \mid x.\mathsf{case}(P,Q) \mid [y \leftrightarrow x] \mid [x \leftarrow N] \end{array}$$

The typing rules for our system are summarized in Fig. 1, which is defined modulo structural congruence. Structural congruence is the least congruence on processes defined by the following rules:

$$\begin{array}{ll} P \mid \mathbf{0} \equiv P & P \equiv_\alpha Q \Rightarrow P \equiv Q \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid Q \equiv Q \mid P \\ x \notin \mathit{fn}(P) \Rightarrow P \mid (\boldsymbol{\nu} x)Q \equiv (\boldsymbol{\nu} x)(P \mid Q) & (\boldsymbol{\nu} x)\mathbf{0} \equiv \mathbf{0} \\ (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P & [y \leftrightarrow x] \equiv [x \leftrightarrow y] \end{array}$$

The operational semantics for the $[y \leftrightarrow x]$ and $[x \leftarrow N]$ constructs, as informed by the proof theory, consist of channel renaming and term substitution, respectively. The channel renaming construct's behavior is to "re-implement" an ambient session on a different name. The reduction rules for our calculus are summarized below:

$$\begin{array}{l} x\langle y\rangle.Q \mid x(z).P \to Q \mid P\{y/z\} \\ x\langle y\rangle.Q \mid !x(z).P \to Q \mid P\{y/z\} \mid !x(z).P \\ x\langle N\rangle.Q \mid x(z).P \to Q \mid P\{N/z\} \\ (\boldsymbol{\nu} x)([x \leftrightarrow y] \mid P) \to P\{y/x\} \\ (\boldsymbol{\nu} x)([x \leftarrow N] \mid P) \to P\{N/x\} \\ x.\mathsf{inl}; P \mid x.\mathsf{case}(Q,R) \to P \mid Q \\ x.\mathsf{inr}; P \mid x.\mathsf{case}(Q,R) \to P \mid R \\ Q \to Q' \Rightarrow P \mid Q \to P \mid Q' \\ P \to Q \Rightarrow (\boldsymbol{\nu} y)P \to (\boldsymbol{\nu} y)Q \\ P \equiv P',\ P' \to Q',\ Q' \equiv Q \Rightarrow P \to Q \end{array}$$

The term substitution construct is similar to the active substitutions of the applied π-calculus, with the particular differences that active substitutions are persistent and applied by structural congruence, while ours obey a linear discipline and are applied by an actual reduction step. Our term language is also very different from the one in the applied π-calculus, since our terms are defined in a functional language that does not include the notion of process calculus (channel) name, whilst the terms in [1] can contain names. A labeled transition system that characterizes relevant external actions can be defined by a judgment $P \xrightarrow{\alpha} Q$, where α denotes an action that can be silent, an output or input of a (bound) name or of a term:

$$\alpha \quad ::= \quad \tau \mid \overline{(\nu z)x\langle z\rangle} \mid x(y) \mid \overline{x\langle N\rangle} \mid x(N)$$

We now present some of the formal results that we have established for our system.

3. Properties of the type system

In this section we establish the results of type preservation and progress for our type system, following the results of [10]. The proof of type preservation relies on several reduction lemmas that relate process reductions with parallel composition through the cut rule. We illustrate these with the cases for the quantifiers.

Lemma 3.1. *Assume*

(a) $\Psi; \Gamma; \Delta_1 \Rightarrow P :: x : \forall y : \tau.A$ *with* $P \overset{x(N)}{\to} P'$

(b) $\Psi; \Gamma; \Delta_2, x : \forall y : \tau.B \Rightarrow Q :: z : C$ *with* $Q \overset{x\langle N\rangle}{\to} Q'$

Then:

(c) $\Psi; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P' \mid Q') :: z : C$

Lemma 3.2. *Assume*

(a) $\Psi; \Gamma; \Delta_1 \Rightarrow P :: x : \exists y : \tau.B$ *with* $P \overset{x\langle N\rangle}{\to} P'$ *and*

(b) $\Psi; \Gamma; \Delta_2, x : \exists y : \tau.B \Rightarrow Q :: z : C$ *with* $Q \overset{x(N)}{\to} Q'$

Then:

(c) $\Psi; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P' \mid Q') :: z : C$

We can now state and sketch the proof of type preservation.

Theorem 3.3 (Type Preservation)**.** *If* $\Psi; \Gamma; \Delta \Rightarrow P :: z : A$ *and* $P \to Q$ *then* $\Psi; \Gamma; \Delta \Rightarrow Q :: z : A$

Proof. By induction on the typing derivation. When the last rule is an instance of cut, we appeal to the reduction lemmas, one for each type C of the cut formula (these are of the form of Lemmas 3.1 and 3.2), or to the rules for renaming and substitution. □

To establish progress, a lemma that establishes a contextual progress property is required. First, we define:

$$\mathit{live}(P) \triangleq P \equiv (\boldsymbol{\nu}\overline{n})(Q \mid R) \quad \text{for some } Q, R, \overline{n}$$

where $Q \equiv \pi.Q'$ (π is a non-replicated prefix), $Q \equiv [x \leftrightarrow y]$ or $Q \equiv [x \leftarrow N]$. Given an action label α, we denote by $s(\alpha)$ the subject of the action α (i.e., the name through which the action takes place). We can now establish the contextual progress property (note the use of $\equiv_S$, defined in Section 2.7).

Lemma 3.4. *Let* $\Psi; \Gamma; \Delta \Rightarrow P :: z : C$. *If* $\mathit{live}(P)$ *then there is* Q *such that one of the following holds:*

(a) $P \to Q$,

(b) $P \overset{\alpha}{\to} Q$ *for some* α *where* $s(\alpha) \in z, \Gamma, \Delta$ *and* $s(\alpha) \in \Gamma, \Delta$ *if* $C = !A$,

(c) $P \equiv_S [x \leftrightarrow z]$, *for some* $x \in \Delta$,

(d) $P \equiv_S [z \leftarrow N]$ *for some* N.

Proof. Induction on typing. The proof is similar to that of [10], with more cases when the last rule applied is cut, to account for renaming, term substitutions, and quantifiers. □

Global progress follows directly from Lemma 3.4.

Theorem 3.5 (Progress)**.** *If* $\cdot; \cdot; \cdot \Rightarrow P :: x : \mathbf{1}$, *and* $\mathit{live}(P)$, *then there exists a process* Q *such that* $P \to Q$.

Note that this is the case because P cannot perform any action α with subject x, since $x : \mathbf{1}$.

The guiding principle mentioned earlier allows us to make a stronger formal connection between cut reductions and pi-calculus reductions, but this is beyond the scope of this particular paper (and is straightforward, given the results of [10] and the earlier presented reductions).

4. Proof irrelevance

We now tackle the problem of eliminating some of the communication overhead generated by the exchange of explicit proof objects. Process calculi are a class of languages that allow us to reason about concurrent processes that may or may not be executing in a distributed setting. If such is indeed the case, there is an argument

$$\frac{}{\Psi;\Gamma;x:A\Rightarrow[x\leftrightarrow z]::z:A}\ \mathsf{id} \qquad \frac{\Psi\vdash M:\tau}{\Psi;\Gamma;\cdot\Rightarrow[z\leftarrow M]::z:\$\tau}\ \$\mathsf{R} \qquad \frac{\Psi,x:\tau;\Gamma;\Delta\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:\$\tau\Rightarrow P::z:C}\ \$\mathsf{L}$$

$$\frac{}{\Psi;\Gamma;\cdot\Rightarrow\mathbf{0}::z:\mathbf{1}}\ \mathbf{1}\mathsf{R} \qquad \frac{\Psi;\Gamma;\Delta\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:\mathbf{1}\Rightarrow P::z:C}\ \mathbf{1}\mathsf{L} \qquad \frac{\Psi;\Gamma;\cdot\Rightarrow P::y:A}{\Psi;\Gamma;\cdot\Rightarrow!z(y).P::z:!A}\ !\mathsf{R}$$

$$\frac{\Psi;\Gamma,u:A;\Delta\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:!A\Rightarrow P\{x/u\}::z:C}\ !\mathsf{L} \qquad \frac{\Psi;\Gamma,u:A;\Delta,y:A\Rightarrow P::z:C}{\Psi;\Gamma,u:A;\Delta\Rightarrow(\boldsymbol{\nu}y)u\langle y\rangle.P::z:C}\ \mathsf{copy}$$

$$\frac{\Psi;\Gamma;\Delta\Rightarrow P::z:A\quad\Psi;\Gamma;\Delta\Rightarrow Q::z:B}{\Psi;\Gamma;\Delta\Rightarrow z.\mathsf{case}(P,Q)::z:A\,\&\,B}\ \&\mathsf{R} \qquad \frac{\Psi;\Gamma;\Delta,x:A\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:A\,\&\,B\Rightarrow x.\mathsf{inl};P::z:C}\ \&\mathsf{L}_1$$

$$\frac{\Psi;\Gamma;\Delta,x:B\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:A\,\&\,B\Rightarrow x.\mathsf{inr};P::z:C}\ \&\mathsf{L}_2 \qquad \frac{\Psi;\Gamma;\Delta_1\Rightarrow P::y:A\quad\Psi;\Gamma;\Delta_2\Rightarrow Q::z:B}{\Psi;\Gamma;\Delta_1,\Delta_2\Rightarrow(\boldsymbol{\nu}y)z\langle y\rangle.(P\mid Q)::z:A\otimes B}\ \otimes\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta,y:A,x:B\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:A\otimes B\Rightarrow x(y).P::z:C}\ \otimes\mathsf{L} \qquad \frac{\Psi;\Gamma;\Delta\Rightarrow P::z:A}{\Psi;\Gamma;\Delta\Rightarrow z.\mathsf{inl};P::z:A\oplus B}\ \oplus\mathsf{R}_1$$

$$\frac{\Psi;\Gamma;\Delta\Rightarrow P::z:B}{\Psi;\Gamma;\Delta\Rightarrow z.\mathsf{inr};P::z:A\oplus B}\ \oplus\mathsf{R}_2 \qquad \frac{\Psi;\Gamma;\Delta,x:A\Rightarrow P::z:C\quad\Psi;\Gamma;\Delta,x:B\Rightarrow Q::z:C}{\Psi;\Gamma;\Delta,x:A\oplus B\Rightarrow x.\mathsf{case}(P,Q)::z:C}\ \oplus\mathsf{L}$$

$$\frac{\Psi,x:\tau;\Gamma;\Delta\Rightarrow P::z:A}{\Psi;\Gamma;\Delta\Rightarrow z(x).P::z:\forall x:\tau.A}\ \forall\mathsf{R} \qquad \frac{\Psi\vdash N:\tau\quad\Psi;\Gamma;\Delta,x:A\{N/y\}\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:\forall y:\tau.A\Rightarrow x\langle N\rangle.P::z:C}\ \forall\mathsf{L}$$

$$\frac{\Psi\vdash N:\tau\quad\Psi;\Gamma;\Delta\Rightarrow P:A\{N/x\}}{\Psi;\Gamma;\Delta\Rightarrow z\langle N\rangle.P::z:\exists x:\tau.A}\ \exists\mathsf{R} \qquad \frac{\Psi,y:\tau;\Gamma;\Delta,x:A\Rightarrow P::z:C}{\Psi;\Gamma;\Delta,x:\exists y:\tau.A\Rightarrow x(y).P::z:C}\ \exists\mathsf{L}$$

$$\frac{\Psi;\Gamma;\Delta_1\Rightarrow P::x:A\quad\Psi;\Gamma;\Delta_2,x:A\Rightarrow Q::z:C}{\Psi;\Gamma;\Delta_1,\Delta_2\Rightarrow(\boldsymbol{\nu}x)(P\mid Q)::z:C}\ \mathsf{cut} \qquad \frac{\Psi;\Gamma;\cdot\Rightarrow P::x:A\quad\Psi;\Gamma,u:A;\Delta\Rightarrow Q::z:C}{\Psi;\Gamma;\Delta\Rightarrow(\nu u)((!u(x).P)\mid Q)::z:C}\ \mathsf{cut}^!$$

Figure 1. A Dependent Type Theory of Sessions.

to be made that trust between the communicating parties should not be assumed outright. In these scenarios, our system, in which properties of the communicated data are ensured by typing but also witnessed by explicit proof objects that are passed by processes, seems to be a reasonable way of addressing the issue of trust (or lack thereof). A client may not trust the remote server code, but provided the server sends the proof objects, the client may in principle check that the proof objects are valid and thus obtains further assurances on the server.

However, it may not necessarily be the case that the communication of explicit proof objects is required by the parties involved. For instance, the properties in question may be easily decidable, or we have a scenario where we have code residing on the same machine that represents multiple communicating sessions (e.g. an operating system, a file system, etc.), or it may be the case that the communicating parties do indeed exist in a distributed setting, but have established trust by some exterior means. In some of these cases we *can* type-check the process code, and so the proof objects are in principle no longer really needed at runtime. Of course, the system as we have presented so far has really no way of determining if it is really the case that a proof object is not used for its computational content. Luckily, proof theory can help us, with the concept of *proof irrelevance* [3, 29].

Proof irrelevance is a technique that allows us to selectively hide portions of a proof. These "hidden" proofs must exist, but it must also be the case that they can be safely erased from a process at runtime. This means that typing must ensure that these hidden proofs are never required to compute something that is not erased. We internalize this notion of proof irrelevance in the functional term language with a new type, $[A]$ (read *bracket* A), meaning that there is a term of type A, but the term itself can be safely erased before runtime without changing the meaning of the process. We can give a precise meaning to $[A]$ by adding a new introduction form for terms, written $[M]$, meaning that M will not be available computationally. We also add a new class of assumptions $x \div A$, meaning that x stands for a term of type A that is not computationally available. Following the style of [29], we define a promotion operation on contexts that transforms computationally irrelevant hypotheses into ordinary ones:

$$\begin{array}{lcl}(\cdot)^{\oplus} & \triangleq & \cdot\\ (\Psi,x:A)^{\oplus} & \triangleq & \Psi^{\oplus},x:A\\ (\Psi,x\div A)^{\oplus} & \triangleq & \Psi^{\oplus},x:A\end{array}$$

We can then define the introduction and elimination forms of proof irrelevant terms:

$$\frac{\Psi^{\oplus}\vdash M:A}{\Psi\vdash[M]:[A]}\ []I \qquad \frac{\Psi\vdash M:[A]\quad\Psi,x\div A\vdash N:C}{\Psi\vdash\mathbf{let}\,[x]=M\,\mathbf{in}\,N:C}\ []E$$

These rules guarantee that a variable of the form $x \div A$ can only be used in terms that are irrelevant (in the technical sense). In such terms, we are allowed to refer to *all* variables, including the irrelevant ones, since the term is not intended to be available at runtime. Terms of bracket type can still be used through the let binding shown above, but the bound variable x is tagged with the irrelevant hypothesis form, to maintain the invariant that no relevant term can use irrelevant variables in a computational manner. Using bracketed types, we ensure that assigned terms are never explored for their computational value, and so can be safely erased at runtime. We first illustrate this with a very simple example and then generalize to our running example of the bank. Consider a very simple process with the following type:

$$\mathsf{T}\triangleq\forall f{:}\mathsf{nat}\to\mathsf{nat}.\forall n{:}\mathsf{nat}.\forall p{:}(n>0).\$\mathsf{nat}\otimes\mathbf{1}$$

The type describes a process that receives a natural number function f, a natural number n and a *proof* that n is strictly positive (for instance, because f is not defined for 0). It will then reply with a natural number (the result of applying f to n) and terminate. A sample process obeying this specification is:

$$\mathsf{Server}\triangleq x(f).x(n).x(p).(\boldsymbol{\nu}y)x\langle y\rangle.([y\leftarrow f(n)]\mid\mathbf{0})::x:\mathsf{T}$$

A sample client that properly interacts with the above process is

$$\mathsf{Client} \triangleq x\langle M\rangle.x\langle 1\rangle.x\langle N\rangle.x(r).[r \leftrightarrow z] :: z : \mathsf{nat}$$

where M must be a term of type $\mathsf{nat} \to \mathsf{nat}$ and N is a term of type $1 > 0$.

Notice that in this situation, the proof object p in Server only serves the purpose of ensuring a restriction on n, its content is never actually used in a computationally meaningful manner. That is, p is a computationally irrelevant proof object. We can now make use of proof irrelevance to identify that the proof object p in Server can be erased at runtime:

$$\mathsf{T_I} \triangleq \forall f : \mathsf{nat} \to \mathsf{nat}.\forall n : \mathsf{nat}.\forall p : [n > 0].\$\mathsf{nat} \otimes \mathbf{1}$$

The server process stays the same, while the Client must now send $[N]$ instead of just N:

$$\mathsf{Client_I} \triangleq x\langle M\rangle.x\langle 1\rangle.x\langle [N]\rangle.x(r).[r \leftrightarrow z] :: z : \mathsf{nat}$$

We can define an operation that, given a well-typed process, erases all terms of bracket type and the respective communication actions. This erasure is obviously not type preserving in general, in the sense that the resulting process may no longer be assigned the same type in our system. However, the erasure is to be applied *after* we have ensured that a process is well typed (and therefore abides by whatever specification is defined in its type), but before the code is actually executed. Thus, the erasure is safe because we know that all properties that typing ensured still hold.

In our example above, the erased server and client processes would be:

$$\begin{array}{lcl} \mathsf{T_e} & \triangleq & \forall f{:}\mathsf{nat} \to \mathsf{nat}.\forall n{:}\mathsf{nat}.\$\mathsf{nat} \otimes \mathbf{1} \\ \mathsf{Server_e} & \triangleq & x(f).x(n).(\boldsymbol{\nu} y)x\langle y\rangle.([y \leftarrow f(n)] \mid \mathbf{0}) :: x : \mathsf{T_e} \\ \mathsf{Client_e} & \triangleq & x\langle M\rangle.x\langle 1\rangle.x(r).[r \leftrightarrow z] :: z : \mathsf{nat} \end{array}$$

The precise definition of the erasure function is standard, since its interaction with the process layer is minimal with the restriction to base types $\$\tau$ only. We therefore elide its formal definition and the companion correctness theorem from this presentation for the sake of brevity.

In our running example of the bank system, if we assume the client trusts the ATM code to not be malicious, we may employ proof irrelevance and write the type of the ATM interface as:

$$\begin{array}{l} \mathsf{TATMClient_I} \triangleq \forall s : \mathsf{string}.\$\mathsf{uid}(s) \multimap \\ \qquad (\forall n : \mathsf{nat}.\$\mathsf{deposit}(s,n) \multimap \exists m : \mathsf{nat}. \\ \qquad \exists p : [\mathsf{n} - 2 \leq \mathsf{m} \leq \mathsf{n}].(\$\mathsf{receipt}(s,m) \otimes 1)) \end{array}$$

which then allows us to safely erase the communication overhead of the proof object p. To conclude, the technique of internalizing proof irrelevance in bracket types provides a clean and modular way of singling out terms (through their types) that are never used for their computational content. This provides us with the opportunity to erase these terms and minimize communication overheads when appropriate.

5. Concluding Remarks

We have presented an interpretation of intuitionistic linear type theory as a dependent session type system for a π-calculus with value passing. Our framework introduces value passing by interpreting the (higher-order) type structure of an underlying functional dependent type theory as atomic from the process perspective. Dependent types may be used to elegantly specify properties of data exchanged by processes in their session types. Previous work [7] encoded these as assertions built into the session type. In particular, we have shown how certified interface contracts, expressing rich properties of distributed protocols, may be expressed in our framework. Our development provides a new account of dependent session types [8] that is completely grounded in logic, and is free from special-purpose technical machinery that is usually required in this setting.

Our approach naturally addresses challenges not yet tackled by other session type systems, such as the use of proof-based certification in scenarios involving communication between untrusted parties. We have also explored proof irrelevance as a way of singling out proofs that may be safely erased at runtime. We have proven that our system ensures type preservation, session fidelity, and global progress. We do not address the issue of describing infinite protocols through recursive types, since the technical challenges of recursive types are well understood and extending our system with recursive types is straightforward and orthogonal to our development.

Several other connections between the π-calculus and linear logic have been established. A first line of research has investigated the use of linearity in type systems (see, e.g., [9, 18, 23, 24]). These type systems have not developed any interpretation of the pure linear logic connectives as behavioral (session) type operators, a program that we have initiated [10], and extend here to the setting of a much richer dependent linear type theory. A second line of work has investigated operational interpretations of linear logic proofs in the π-calculus and related models (see, e.g., [2, 5, 6, 21]). We may broadly characterize these as applications of the π-calculus as a convenient language for analyzing linear logic proof objects, while our aim is to develop the linear propositions-as-types paradigm as a foundation for distributed, session-based, practical programming languages, with rich interface specifications.

In future work, we plan on extending our program of providing logical explanations to the phenomena of concurrency to multiparty session types, which are a generalization of the binary session types we have given logical meaning in this and prior work. To achieve this, we plan to investigate potential relationships of multi-party sessions to linear modal logic [16], which provides a natural way of reasoning about several principals. Another interesting line of research is the development of appropriate theories of bisimulation and observational equivalence for (dependent) session types and the study of their relationship to forms of logical and proof equivalence. Finally, we also wish to consider a potentially tighter integration of functional and concurrent computation that does not require the two-layer stratification that we have presented in this paper. Ongoing research in concurrent evaluation strategies for functional programs using logical interpretations might provide deeper insights in this particular direction.

Acknowledgments

Support for this research was provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program, under grants SFRH / BD / 33763 / 2009 and INTERFACES NGN-44 / 2009, and CITI.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages*, POPL'01, pages 104–115. ACM, 2001.

[2] S. Abramsky. Computational Interpretations of Linear Logic. *Theor. Comp. Sci.*, 111(1&2), 1993.

[3] S. Awodey and A. Bauer. Propositions as [types]. *J. Log. Comput.*, 14(4):447–471, 2004.

[4] A. Barber and G. Plotkin. Dual Intuitionistic Linear Logic. Technical Report LFCS-96-347, Univ. of Edinburgh, 1997.

[5] E. Beffara. A Concurrent Model for Linear Logic. *ENTCS*, 155:147–168, 2006.

[6] G. Bellin and P. Scott. On the π-Calculus and Linear Logic. *Theor. Comp. Sci.*, 135:11–65, 1994.

[7] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *21st International Conference on Concurrency Theory*, CONCUR'10, pages 162–176. Springer LNCS 6269, 2010.

[8] E. Bonelli, A. Compagnoni, and E. L. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. of Func. Prog.*, 15(2):219–247, 2005.

[9] L. Caires. Logical Semantics of Types for Concurrency. In *International Conference on Algebra and Coalgebra in Computer Science*, CALCO'07, pages 16–35. Springer LNCS 4624, 2007.

[10] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory*, CONCUR'10, pages 222–236. Springer LNCS 6269, 2010.

[11] L. Caires and H. T. Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, 2010.

[12] I. Cervesato and F. Pfenning. A linear logical framework. *Inf. & Comput.*, 179(1), 2002.

[13] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[14] T. Coquand and G. Huet. The calculus of constructions. *Inf. & Comput.*, 76:95–120, February 1988.

[15] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys 2006*, pages 177–190. ACM, 2006.

[16] D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. A linear logic of affirmation and knowledge. In *Proceedings of the 11th European Symposium on Research in Computer Security*, ESORICS'06, pages 297–312. Springer LNCS 4189, Sept. 2006.

[17] S. Gay and M. Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[18] M. Giunti and V. T. Vasconcelos. A Linear Account of Session Types in the Pi-Calculus. In *21st International Conference on Concurrency Theory*, CONCUR'10, pages 432–446. Springer LNCS 6269, 2010.

[19] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40:143–184, January 1993.

[20] K. Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[21] K. Honda and O. Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comp. Sci.*, 411:2223–2238, 2010.

[22] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems*, ESOP'98, pages 122–138. Springer LNCS 1381, 1998.

[23] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *28th Symposium on Principles of Programming Languages*, POPL'01, pages 128–141. ACM, 2001.

[24] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *23rd Symposium on Principles of Programming Languages*, POPL'96, pages 358–371. ACM, 1996.

[25] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[26] R. Milner. Functions as processes. *Math. Struct. in Comp. Sci.*, 2(2):119–141, 1992.

[27] G. C. Necula. Proof-carrying code. In *24th Symposium on Principles of Programming Languages*, POPL'97, pages 106–119. ACM, 1997.

[28] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[29] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Symposium on Logic in Computer Science*, LICS'01, pages 221–230. IEEE Computer Society, 2001.

[30] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[31] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation*, PLDI'98, pages 249–257. ACM, 1998.