Functions as Session-Typed Processes

Bernardo Toninho[1,2], Luis Caires[2], and Frank Pfenning[1]

[1] Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
[2] CITI and Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Lisboa, Portugal

Abstract. We study type-directed encodings of the simply-typed λ-calculus in a session-typed π-calculus. The translations proceed in two steps: standard embeddings of simply-typed λ-calculus in a linear λ-calculus, followed by a standard translation of linear natural deduction to linear sequent calculus. We have shown in prior work how to give a Curry-Howard interpretation of the proofs in the linear sequent calculus as π-calculus processes subject to a session type discipline. We show that the resulting translations induce sharing and copying parallel evaluation strategies for the original λ-terms, thereby providing a new logically motivated explanation for these strategies.

1 Introduction

The goal of this paper is to study type-directed encodings of the simply-typed λ-calculus in a session-typed π-calculus, including a novel encoding that captures a reduction strategy in which shared sub-expressions are evaluated in parallel (providing a logical justification for *futures* [10]). Unlike other proposals, our encodings are canonically extracted from standard translations of linear natural deduction to linear sequent calculus in standard logical systems, based on the interpretation of session-typed processes as intuitionistic linear logic proofs we developed in [6].

Milner [17] presented two translations of the λ-calculus into the π-calculus: one capturing call-by-name and one capturing call-by-value. This provided evidence for the universality of the π-calculus for expressing sequential computation. Since then, a number of other translations have been developed (see [19]).

An interesting foundational question is if similar translations exist from *typed* λ-calculus to a *typed* π-calculus. Van Bakel and Vigliotti [21] give a partial answer by providing a conceptually different translation into the *asynchronous* π-calculus where the image of simply-typed λ-terms can be assigned types in Gentzen's classical sequent calculus LK. In this paper we provide another answer, also rooted in logic. We first translate from the simply-typed λ-calculus to a *linear* λ-calculus, using either of the two canonical embeddings proposed by Girard [8] as detailed by Maraist et al [15]. The linear λ-calculus is related to intuitionistic linear logic by a Curry-Howard isomorphism [22, 1], so we can now apply a standard translation from natural deduction to the dual intuitionistic linear sequent calculus. In prior work [6], we have shown how

to view this sequent calculus as a type assignment system for the π-calculus, capturing *session types* [12] in a purely logical manner. Viewed end-to-end, we have two type-directed translations from the simply-typed λ-calculus to the session-typed π-calculus, depending which of Girard's embeddings of intuitionistic logic in linear logic we use.

Remarkably, the translations constructed in this type-directed manner are closely related to Milner's two original translations that capture call-by-name and call-by-value. While our call-by-name translation behaves the same way, our analog of the call-by-value translation is subtly different in that the resulting terms behave like *futures* [10]: in an application $(\lambda x.\, M)\, N$ reductions in the function body M and the argument N can proceed in parallel, synchronizing only when the value of x is needed. If we always proceed with reducing N first, we obtain call-by-value. If we always proceed with the body M first we obtain call-by-need. So perhaps it is more appropriate to characterize the semantic endpoints of the two translations as *copying* (as in call-by-name) and *sharing* (as in futures, relaxing the sequentiality constraints of call-by-value and call-by-need). Remarkably, the sharing semantics thus obtained is reminiscent of sharing reductions as studied in the context of optimal reduction strategies [14, 9], a study that goes back to Wadsworth's thesis [23], although in our case it just comes out naturally from our logically grounded translations. Furthermore, to the best of the authors' knowledge, such a sharing encoding of the λ-calculus in the π-calculus was not identified before this work.

Another remarkable property, due to the simple logical nature of the interpretations, is that the linear target type of the translation is *inhabited* by a π-calculus term if and only if the source type is inhabited by a λ-calculus term In a sense, the linear session type discipline is a tight characterization for simple typing in the λ-calculus source, exposing intrinsic parallelism in its structure. We thus show that session types are powerful enough to type functional evaluation.

In the remainder of this paper we proceed as follows: We detail our translation of the λ-calculus to the π-calculus, by presenting the several intermediate steps (Sections 2.1, 2.2) and composing them in Section 2.4. In Section 3 we show the operational soundness and completeness of our translation. Finally, we present some concluding remarks and a discussion of related work in Section 4.

2 From the λ-calculus to the π-calculus

In this section we present several translations which, in combination, allow us to translate typed λ-calculus terms to *session-typed* π-calculus processes. We first review two well-known translations from the simply typed λ-calculus to the linear λ-calculus, introduced in Girard's seminal paper [8] as translations from intuitionistic to linear logic, and hence by the Curry-Howard correspondence, from the λ-calculus to the linear λ-calculus [15, 1]. We then show how to translate from the linear λ-calculus (logically, a system of natural deduction) to a sequent formulation of linear logic. The latter will yield the desired π-calculus terms by a Curry-Howard interpretation.

2.1 Interpreting the λ-calculus in the linear λ-calculus

We consider the simply typed λ-calculus with function types $T \to S$ and base types b. Our translation naturally extends to products and sums, interpreted as the product and

sum types of the linear λ-calculus, but we refrain from presenting those here for the sake of simplicity. The simply-typed λ calculus is given by the following grammar of terms M, N and types T, S, where b denotes base types. The typing rules for the calculus are given below, inductively defining the judgment $\Gamma \vdash M : T$, where Γ denotes a finite set of unique typing assumptions of the form $x : T$, as usual.

Definition 2.1 (λ-calculus Terms and Typing).

$$\begin{array}{rl} M, N & ::= x \mid \lambda x : T.\, M \mid M\, N \\ T, S & ::= T \to S \mid b \end{array}$$

$$\frac{}{\Gamma, x : T \vdash x : T}\ \mathsf{hyp} \qquad \frac{\Gamma, x : T \vdash M : S}{\Gamma \vdash \lambda x : T.\, M : T \to S} \to I$$

$$\frac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash M\, N : T} \to E$$

The linear λ-calculus is a more fine-grained version of the λ-calculus, in which variables in a λ-abstraction must be used exactly once. The calculus is endowed with a modal type $!T$ (referred to as an exponential) that allows for unrestricted usage of variables, in order to obtain the full generality of the λ-calculus. We distinguish between the unrestricted and linear λ-abstraction by using $\hat{\lambda}$ for the latter. The linear function type is written as $\multimap$. The terms and types of the linear λ-calculus are given below.

Definition 2.2 (Linear λ-calculus Terms).

$$\begin{array}{rl} M, N & ::= x \mid u \mid \hat{\lambda} x : T.M \mid M\, N \mid\, !M \mid \mathsf{let}\ !u = M\ \mathsf{in}\ N \\ T, S & ::= T \multimap S \mid\, !T \mid b \end{array}$$

We syntactically distinguish between linear variables x and unrestricted variables u (typing ensures linear variables are used exactly once, while unrestricted variables can be used arbitrarily often). The terms $!M$ and $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ are the introduction and elimination forms of the exponential type, respectively, where the variable u can occur in the term N in an unrestricted manner.

The typing rules for the linear calculus are given below, defining the judgment $\Gamma; \Delta \vdash M : T$ with Δ denoting *linear* (used exactly once) assumptions of the form $x : T$ (not subject to weakening or contraction) and Γ unrestricted assumptions $u : T$.

Definition 2.3 (Linear λ-calculus Typing).

$$\frac{}{\Gamma; x : T \vdash x : T}\ \mathsf{hyp} \qquad \frac{}{\Gamma, u : T; \cdot \vdash u : T}\ \mathsf{uhyp}$$

$$\frac{\Gamma; \Delta, x : T \vdash M : S}{\Gamma; \Delta \vdash \hat{\lambda} x : T.M : T \multimap S} \multimap I \qquad \frac{\Gamma; \cdot \vdash M : T}{\Gamma; \cdot \vdash\, !M : !T}\ !I$$

$$\frac{\Gamma; \Delta \vdash M : T \multimap S \quad \Gamma; \Delta' \vdash N : T}{\Gamma; \Delta, \Delta' \vdash M\, N : S} \multimap E$$

$$\frac{\Gamma; \Delta \vdash M : !T \quad \Gamma, u : T; \Delta' \vdash N : S}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ !u = M\ \mathsf{in}\ N : S}\ !E$$

Given that the Curry-Howard correspondence allows us to identify the simply typed λ-calculus with intuitionistic logic and the linear λ-calculus with linear intuitionistic logic, we make use of two well-known embeddings of intuitionistic logic in linear logic to interpret the λ-calculus in the linear λ-calculus. These embeddings originate from Girard's seminal paper on linear logic [8] and are detailed by Maraist et al [15]. The first translation, which we will refer to as the $[\cdot]$ translation, is defined inductively on types, terms and contexts below.

Definition 2.4 (The $[\cdot]$ Translation).

$$
\begin{array}{ll}
[T \to S] & \triangleq (![T]) \multimap [S] \\
[b] & \triangleq b \\
 & \\
[x] & \triangleq u_x \\
[\lambda x : T.\, M] & \triangleq \hat{\lambda} x : ![T].\, \mathsf{let}\ !u_x = x\ \mathsf{in}\ [M] \\
[M\ N] & \triangleq [M]\ (![N]) \\
 & \\
[\Gamma, x : T] & \triangleq [\Gamma], x : [T]
\end{array}
$$

Intuitively, this translation maps λ-abstractions to linear λ-abstractions, where the argument is forced to be of exponential type, and thus can be let-bound to a unrestricted variable u_x. We write u_x for a fresh unrestricted variable encoding the (linear) variable x. The translation of application enforces the invariant, resulting in an application of the translated terms where the argument is prefixed with !. The correctness of the translation can be shown by a straightforward induction on typing.

Theorem 2.5 (Correctness of the $[\cdot]$ Translation). *If $\Gamma \vdash M : T$ then $[\Gamma]; \cdot \vdash [M] : [T]$.*

The second translation, which we denote as the $(\cdot)^*$ translation (of which Girard curiously remarked: "*This boring translation is reminiscent of the modal translation of intuitionistic logic*") is slightly more involved, being inductively defined using an auxiliary translation, denoted by $(\cdot)^+$. The idea behind this translation is that all components of composite types are prefixed with a !, which results in the following inductive definition on types, terms and contexts, given below.

Definition 2.6 (The $(\cdot)^*$ Translation).

$$
\begin{array}{ll}
(T)^* & \triangleq\ !T^+ \\
(T \to S)^+ & \triangleq T^* \multimap S^* \\
(\tau)^+ & \triangleq \tau \\
 & \\
(x)^* & \triangleq\ !u_x \\
(\lambda x : T.\, M)^* & \triangleq\ !(\hat{\lambda} x : !T^+.\, \mathsf{let}\ !u_x = x\ \mathsf{in}\ M^*) \\
(M\ N)^* & \triangleq (\mathsf{let}\ !u = M^*\ \mathsf{in}\ u)\ N^* \\
 & \\
(\Gamma, x : T)^+ & \triangleq (\Gamma)^+, x : T^+
\end{array}
$$

Similar to the previous translation, we can show the correctness of the translation through a straightforward induction on typing.

Theorem 2.7 (Correctness of the $(\cdot)^*$ Translation). *If $\Gamma \vdash M : T$ then $\Gamma^+; \cdot \vdash M^* : T^*$ and $\Gamma^+; \cdot \vdash M^+ : T^+$.*

Using the two translations given above, we can take any well-typed λ-calculus term and translate it to a corresponding well-typed linear λ-calculus term (we omit type annotations in abstractions for readability purposes). For instance, the redex $(\lambda x.M)\, N$ can be translated into the following linear terms (we denote by $\longrightarrow$ the linear λ calculus operational semantics, as given in Def. 3.1 – by-name application, by-value let binding, where the values are λ-abstractions and terms of the form $!M$):

$$\begin{array}{rcl}
[(\lambda x.\, M)\, N] & = & (\hat{\lambda} x.\, \mathsf{let}\ !u_x = x\ \mathsf{in}\ [M])\, (![N]) \\
& \longrightarrow & \mathsf{let}\ !u_x = ![N]\ \mathsf{in}\ [M] \\
& \longrightarrow & [M]\{[N]/u_x\} \\
((\lambda x.M)\, N)^* & = & (\mathsf{let}\ !u = !(\hat{\lambda} x.\, \mathsf{let}\ !u_x = x\ \mathsf{in}\ M^*)\ \mathsf{in}\ u)\, N^* \\
& \longrightarrow & (\hat{\lambda} x.\, \mathsf{let}\ !u_x = x\ \mathsf{in}\ M^*)\, N^* \\
& \longrightarrow & \mathsf{let}\ !u_x = N^*\ \mathsf{in}\ M^*
\end{array}$$

As can be seen in the example reductions above, the $[\cdot]$ translation induces a call-by-name reduction strategy on λ terms, while the $(\cdot)^*$ translation induces call-by-value. These observations are made precise by Maraist et al. [15].

Our goal is to ultimately translate a λ-calculus term to a π-calculus process, which as we detail in the following sections, is achieved by translating a linear λ-calculus term into a sequent calculus proof and then interpreting the result as a session-typed process.

2.2 From Natural Deduction to Sequent Calculus

The Curry-Howard correspondence of linear logic and the linear λ-calculus allows us to move interchangeably between typing derivations and linear logic proofs in natural deduction form. Another typical form of presenting logic is through a sequent calculus. A sequent calculus consists of a collection of so-called left and right rules which define how to use and prove a particular proposition, respectively. We can further equip a sequent calculus with a faithful proof term assignment, where proof terms serve as compact notation for proofs. The rules for a sequent calculus for linear logic (in particular, the fragment with implication and exponential) are given below, defining the judgment $\Gamma; \Delta \Rightarrow D : A$, meaning that D is a proof term for the proposition A, under the linear assumptions recorded in Δ and the unrestricted assumptions recorded in Γ. This formulation is usually called Dual Intuitionistic Linear Logic (DILL) [1].

Definition 2.8 (Sequent Calculus for DILL).

$$\frac{}{\Gamma; x : A \Rightarrow \mathsf{id}\ x : A}\ \mathsf{id} \qquad \frac{\Gamma; \Delta, x : A \Rightarrow D : B}{\Gamma; \Delta \Rightarrow \multimap\mathsf{R}\ (x.\, D) : A \multimap B}\ \multimap\mathsf{R}$$

$$\frac{\Gamma; \cdot \Rightarrow D : A}{\Gamma; \cdot \Rightarrow !\mathsf{R}\ D : !A}\ !\mathsf{R} \qquad \frac{\Gamma; \Delta \Rightarrow D : A \quad \Gamma; \Delta', y : B \Rightarrow E : C}{\Gamma; \Delta, \Delta', x : A \multimap B \Rightarrow \multimap\mathsf{L}\ x\ D\ (y.\, E) : C}\ \multimap\mathsf{L}$$

$$\frac{\Gamma, u : A; \Delta \Rightarrow D : C}{\Gamma; \Delta, x : !A \Rightarrow !\mathsf{L}\ x\ (u.\, D) : C}\ !\mathsf{L} \qquad \frac{\Gamma, u : A; \Delta, x : A \Rightarrow D : C}{\Gamma, u : A; \Delta \Rightarrow \mathsf{copy}\ u\ (x.\, D) : C}\ \mathsf{copy}$$

$$\frac{\Gamma; \Delta \Rightarrow D : A \quad \Gamma; \Delta', x : A \Rightarrow E : C}{\Gamma; \Delta, \Delta' \Rightarrow \mathsf{cut}\ D\ (x.\, E) : C}\ \mathsf{cut}$$

$$\frac{\Gamma; \cdot \Rightarrow D : A \quad \Gamma, u : A; \Delta \Rightarrow E : C}{\Gamma; \Delta \Rightarrow \mathsf{cut}^!\ D\ (u.\, E) : C}\ \mathsf{cut}^!$$

It is possible to translate proofs in natural deduction style to sequent calculus, and therefore by the Curry-Howard correspondence we can straightforwardly translate linear λ-terms to the sequent calculus proof terms given above. Intuitively, the introduction forms ($\hat{\lambda}$ abstractions and !) are directly translated to the proof terms corresponding to the respective right rules. The elimination forms (application and let) are translated to instances of cut with their respective left rules. This translation is defined below, written as $[\![\cdot]\!]$, and is the computational content of the following theorem (modulo α-equivalence).

Theorem 2.9. *If $\Gamma; \Delta \vdash M : A$ then $\Gamma; \Delta \Rightarrow [\![M]\!] : A$.*

Definition 2.10 (The $[\![\cdot]\!]$ Sequent Calculus Translation).

$$\begin{array}{ll}
[\![x]\!] & \triangleq \mathsf{id}\ x \\
[\![u]\!] & \triangleq \mathsf{copy}\ u\ (x.\, \mathsf{id}\ x) \\
[\![\hat{\lambda}x.M]\!] & \triangleq \multimap\mathsf{R}\ (x.\, [\![M]\!]) \\
[\![M\ N]\!] & \triangleq \mathsf{cut}\ [\![M]\!]\ (x.\, \multimap\mathsf{L}\ x\ [\![N]\!]\ (y.\, \mathsf{id}\ y)) \\
[\![!M]\!] & \triangleq !\mathsf{R}\ M \\
[\![\mathsf{let}\ !u = M\ \mathsf{in}\ N]\!] & \triangleq \mathsf{cut}\ [\![M]\!]\ (x.\, !\mathsf{L}\ x\ (u.\, [\![N]\!]))
\end{array}$$

The relevance of this translation step is made clear in the next section, where we detail a tight correspondence between sequent calculus proofs and session-typed processes in a π-calculus.

There is also a more complex translation from natural deduction to sequent calculus, which eliminates some unnecessary instances of cut. An instance cut in a proof corresponds to a process reduction and so this other translation can be seen as a more optimized version, with fewer administrative reduction steps.

2.3 Linear logic and session types

It turns out that there exists a tight correspondence between linear logic proofs and session typed π-calculus processes [6, 20]. We summarize it here in the following rules for

the fragment of linear logic we are interested in. Letters P, Q, R range over processes, as defined below.

Definition 2.11 (π-calculus Processes).

$$P, Q ::= \mathbf{0} \mid P|Q \mid (\nu y)P \mid x\langle y\rangle.P \mid x(y).P \mid !x(y).P \mid [y \leftrightarrow x]$$

The process calculus is mostly standard. $\mathbf{0}$ stands for the terminated process, $P|Q$ is the parallel composition of processes P and Q, $(\nu y)P$ denotes the binding of a channel name y whose scope is P, $x\langle y\rangle.P$ denotes the process that outputs y along channel x and continues as P, $x(y).P$ stands for the process that inputs along x and binds the received value to y in P, $!x(y).P$ denotes replicated input and $[x \leftrightarrow y]$ is a channel forwarding construct, that equates channel names x and y [20]. We consider it here as a primitive construct, even if in the typed language it may be encoded by a copycat process generated by expanding proofs of non-atomic identity axioms as shown in [6], in a way closely related to the internal mobility translation of [18].

The operational semantics of the processes given above are defined modulo a structural congruence relation, written $P \equiv Q$.

Definition 2.12 (Structural Congruence). *Structural congruence is the least congruence satisfying the following rules*

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

All structural congruence rules given above are standard. The reduction rules for processes are as follows.

Definition 2.13 (Operational Semantics of Processes).

$$\begin{array}{ll}
x\langle y\rangle.Q \mid x(z).P \to Q \mid P\{y/z\} & \\
x\langle y\rangle.Q \mid !x(z).P \to Q \mid P\{y/z\} \mid !x(z).P & \\
(\nu x)([x \leftrightarrow y] \mid P) \to P\{y/x\} & \textit{(provided } x \neq y\textit{)} \\
Q \to Q' \Rightarrow P \mid Q \to P \mid Q' & \\
P \to Q \Rightarrow (\nu y)P \to (\nu y)Q & \\
P \equiv P',\ P' \to Q',\ Q' \equiv Q \Rightarrow P \to Q &
\end{array}$$

The typing judgment for processes is $\Gamma; \Delta \Rightarrow P :: z : A$, where P is a process that implements a session of type A on channel z, when composed with processes implementing the sessions in Γ and Δ. All channel names in Γ, Δ and z must be mutually

distinct, and we may implicitly rename bound names to maintain this invariant.

$$\frac{}{\Gamma; x : A \Rightarrow [x \leftrightarrow z] :: z : A}\ \mathsf{id} \qquad \frac{\Gamma; \Delta, x : A \Rightarrow P :: z : B}{\Gamma; \Delta \Rightarrow z(x).P :: z : A \multimap B}\ \multimap\mathsf{R}$$

$$\frac{\Gamma; \cdot \Rightarrow P :: x : A}{\Gamma; \cdot \Rightarrow\ !z(x).P :: z : !A}\ !\mathsf{R} \qquad \frac{\Gamma, u : A; \Delta \Rightarrow P :: z : C}{\Gamma; \Delta, x : !A \Rightarrow P\{x/u\} :: z : C}\ !\mathsf{L}$$

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

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

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

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

$A \multimap B$ corresponds to the session input type (i.e. input a session of type A and proceed as B), $!A$ corresponds to a persistent session of type A, while cut corresponds to session composition (and cut elimination to reduction). The left rules indicate how to use a session of a given type. The translation of a sequent calculus proof term D to a well-typed process $\hat{D}^z$ is a straightforward mapping of the sequent calculus rules to the appropriate typing rules, singling out a distinguished channel name z.

Definition 2.14 (Translation from Sequent to Type Derivations). *The translation of a proof term D to a process $\hat{D}^z$, written $D \rightsquigarrow \hat{D}^z$ is defined by:*

$$\begin{array}{ll}
\mathsf{id}\ x & \rightsquigarrow [x \leftrightarrow z] \\
\multimap\mathsf{R}\ (y.\, D) & \rightsquigarrow z(y).\hat{D}^z \\
\multimap\mathsf{L}\ x\ D\ (x.\, E) & \rightsquigarrow (\nu y)x\langle y\rangle.(\hat{D}^y \mid \hat{E}^z) \\
!\mathsf{R}\ D & \rightsquigarrow\ !z(y).\hat{D}^y \\
!\mathsf{L}\ x\ (u.\, D) & \rightsquigarrow \hat{D}^z\{x/u\} \\
\mathsf{copy}\ u\ (y.\, D) & \rightsquigarrow (\nu y)u\langle y\rangle.\hat{D}^z \\
\mathsf{cut}\ D\ (x.\, E) & \rightsquigarrow (\nu x)(\hat{D}^x \mid \hat{E}^z) \\
\mathsf{cut}^!\ D\ (u.\, E) & \rightsquigarrow (\nu u)((!u(y).\hat{D}^y) \mid \hat{E}^z)
\end{array}$$

The correctness of the translation, and other results regarding the correspondence mentioned above, including some discussion on channel linking are detailed in [6].

Finally, by composing the $[\![\cdot]\!]$ and process translations, we can translate a linear λ-calculus term to a process (we refer to this composition as $[\![\cdot]\!]_z$, where z is the name on which the process implements the appropriate session).

Definition 2.15 (The $[\![\cdot]\!]_z$ Translation).

$$
\begin{array}{ll}
[\![x]\!]_z & \triangleq [x \leftrightarrow z] \\
[\![u]\!]_z & \triangleq (\nu x)u\langle x\rangle.[x \leftrightarrow z] \\
[\![\hat{\lambda}x.M]\!]_z & \triangleq z(x).[\![M]\!]_z \\
[\![M\ N]\!]_z & \triangleq (\nu x)([\![M]\!]_x \mid (\nu y)x\langle y\rangle.([\![N]\!]_y \mid [x \leftrightarrow z])) \\
[\![!M]\!]_z & \triangleq\ !z(x).[\![M]\!]_x \\
[\![\mathsf{let}\ !u = M\ \mathsf{in}\ N]\!]_z & \triangleq (\nu x)([\![M]\!]_x \mid [\![N]\!]_z\{x/u\})
\end{array}
$$

And thus we have the following correctness theorem, which follows straightforwardly from the composition of Theorems 2.9 and the correctness of the process translation.

Theorem 2.16. *If* $\Gamma; \Delta \vdash M : A$ *then* $\Gamma; \Delta \Rightarrow [\![M]\!]_z :: z : A$

2.4 Composing the translations

We can now compose the translations from Sections 2.1 and 2.2 with the translation $[\![\cdot]\!]_z$ defined above to give translations from the λ-calculus to the session typed process calculus of the previous section. Note that while our typing discipline requires channel linking (arising from the proof theory), this is done without loss of generality, given that it is possible to encode this behavior using a forwarding process [18, 4].

Copying Translation Let us first consider the composition of $[\cdot]$ with $[\![\cdot]\!]_z$, which we will write as $[\cdot]_z$ (we will omit the translation of types and contexts since those are the same as in $[\cdot]$):

$$
\begin{array}{ll}
[x]_z & \triangleq (\nu x)u_x\langle x\rangle.[x \leftrightarrow z] \\
[\lambda x.M]_z & \triangleq z(x).(\nu y)([x \leftrightarrow y] \mid [M]_z\{y/u_x\}) \\
[M\ N]_z & \triangleq (\nu w)([M]_w \mid (\nu y)w\langle y\rangle.((!y(x).[N]_x) \mid [w \leftrightarrow z]))
\end{array}
$$

By composing the translation correctness theorems (Theorems 2.5 and 2.16) we obtain the following correctness result.

Theorem 2.17. *If* $\Gamma \vdash M : T$ *then* $[\Gamma]; \cdot \Rightarrow [M]_z :: z : [T]$.

It is known that the $[\cdot]$ translation from the λ-calculus to the linear λ-calculus corresponds to a call-by-name evaluation strategy. We can observe the evaluation strategy induced by our $[\cdot]_z$ translation by considering the process in the image of the translation of a β-redex (we write $\rightarrow^n$ for the n-fold iteration of $\rightarrow$):

$$
\begin{array}{rl}
[(\lambda x.M)\ N]_z = & (\nu w)(w(x).(\nu y')([x \leftrightarrow y'] \mid [M]_w\{y'/u_x\}) \\
& \quad \mid (\nu y)w\langle y\rangle.((!y(x).[N]_x) \mid [w \leftrightarrow z])) \\
\rightarrow & (\nu w)(\nu y)(\nu y')([y \leftrightarrow y'] \mid [M]_w\{y'/u_x\} \\
& \quad \mid\ !y(x).[N]_x \mid [w \leftrightarrow z]) \\
\rightarrow^2 & (\nu y)([M]_z\{y/u_x\} \mid\ !y(x).[N]_x)
\end{array}
$$

As we can see above, in the $[\cdot]_z$ translation, a β-redex is translated to a process that is the parallel composition of the translation of the body M of the λ-abstraction and a

replicated instance of the argument N (both sharing a private channel y, along which they can communicate). For each occurrence of the variable x in M, the translation will generate outputs along y that trigger the evaluation of a new copy of N. For this reason, we call the evaluation strategy induced by $\lceil\cdot\rceil_z$ a *copying* evaluation strategy.

This translation is surprisingly similar to that originally presented by Milner in [17] as the call-by-name translation of the λ-calculus. Milner's untyped translation is (we use a different notation than Milner's since it overlaps with our $[\![\cdot]\!]$ translation):

$$\begin{array}{ll} \lceil x\rceil_z & \triangleq x\langle z\rangle \\ \lceil \lambda x.M\rceil_z & \triangleq z(x).z(v).\lceil M\rceil_v \\ \lceil M\ N\rceil_z & \triangleq (\nu w)(\lceil M\rceil_w \mid (\nu y)w\langle y\rangle.w\langle z\rangle.(!y(x).\lceil N\rceil_x)) \end{array}$$

Milner's additional communication steps fundamentally play the role of our channel links. In the translation of a variable, we output a fresh name which we then link to the free name z, while Milner's outputs z outright. In the translation of application, there is an output of a fresh name y (that will be used by the function to receive its argument) and then an output of the free name z, which is the distinguished channel of the translation. We avoid this second output by, after sending the fresh channel that will be used to communicate with the argument, linking the name w with the distinguished name z. In essence, Milner's translation is fundamentally the same as ours (modulo some minor syntactical issues). As mentioned above, it is possible to eliminate channel linking in the translation by replacing the explicit linking construct with a forwarder process in the style of [18, 4], or by appealing to a natural notion of observational equivalence. Nonetheless, our typing discipline requires the channel links, thus typing modulo this notion of observational equivalence is left for future work. The fact that his untyped translation, in some sense, predicted the one that can be derived by proof theory, further acknowledges Milner's foresight.

However, we are motivated by the proof theoretical underpinnings of type preserving translations, which is in some sense a more canonical approach to the problem. Furthermore, Milner's goal was to encode call-by-name and call-by-value as closely as possible, using the π-calculus. Our approach just examines the result of standard proof-theoretic translations and observes that they are distinguished not so much by evaluation order, but by the degree of sharing.

Sharing Translation Similarly, we can compose $(\cdot)^*$ with $[\![\cdot]\!]_z$, written $[\![\cdot]\!]^*_z$, as follows:

$$\begin{array}{ll} [\![x]\!]^*_z & \triangleq\ !z(a).(\nu x)u_x\langle x\rangle.[x \leftrightarrow a] \\ [\![\lambda x.M]\!]^*_z & \triangleq\ !z(a).a(x).(\nu y)([x \leftrightarrow y] \mid [\![M]\!]^*_a\{y/u_x\}) \\ [\![M\ N]\!]^*_z & \triangleq\ (\nu w)((\nu x)([\![M]\!]^*_x \mid (\nu v)x\langle v\rangle.[v \leftrightarrow w]) \mid \\ & \quad\ (\nu y)w\langle y\rangle.([\![N]\!]^*_y \mid [w \leftrightarrow z])) \end{array}$$

As for the previous translation, we can compose Theorems 2.7 and 2.16 to obtain the correctness of the translation.

Theorem 2.18. *If* $\Gamma \vdash M : A$ *then* $\Gamma^+; \cdot \Rightarrow [\![M]\!]^*_z :: z : A^*$.

The $(\cdot)^*$ translation from λ to linear λ terms corresponds to call-by-value. However, let us consider the translation of a β-redex in the $[\![\cdot]\!]_z$ translation:

$$\begin{aligned}
[\![(\lambda x.M)\,N]\!]^*_z = \;& (\nu w)((\nu x)(!x(a).a(b).(\nu y')([b \leftrightarrow y'] \mid [\![M]\!]^*_a\{y'/u_x\}) \\
& \mid (\nu v)x\langle v\rangle.[v \leftrightarrow w]) \mid (\nu y)w\langle y\rangle.([\![N]\!]^*_y \mid [w \leftrightarrow z])) \\
\rightarrow^2 \;& (\nu w)(\nu y)(w(b).(\nu y')([b \leftrightarrow y'] \mid [\![M]\!]^*_w\{y'/u_x\}) \\
& \mid w\langle y\rangle.([\![N]\!]^*_y \mid [w \leftrightarrow z])) \\
\rightarrow^3 \;& (\nu y)([\![M]\!]^*_z\{y/u_x\} \mid [\![N]\!]^*_y)
\end{aligned}$$

In the redex above, we obtain the translation of the abstraction body M in parallel with the translation of the argument N (sharing the private name y for communication). Unlike in the copying reduction strategy, the reductions of the argument and the abstraction body can proceed concurrently, generating no extra copies of N (note that this translation explicitly guards λ-abstractions and variables with replication). Since the reductions of N are *shared* by all variable occurrences in a λ-abstraction, we call this a *sharing* evaluation strategy. This strategy is that of *futures* in Multilisp [10], in which the body of a function can be evaluated in parallel with its argument, synchronizing upon the parameter variable.

One interesting fact about this viewpoint of *sharing vs copying*, is that in the sharing translation, we obtain what can be understood as call-by-value and call-by-need as two extremal execution traces of the resulting processes (i.e., on one extreme we reduce a function argument all the way to a value, and only after that we begin to reduce the abstraction body – call-by-value; on the other extreme we never reduce the argument until it is actually needed by the abstraction body – call-by-need).

As was previously mentioned, the $(\cdot)^*$ translation was discovered to induce a call-by-value reduction strategy at the linear λ-calculus level [15]. One may then wonder why our $[\![\cdot]\!]^*_z$ translation (which makes use of $(\cdot)^*$) is "looser", in some sense, given that it does not force arguments to reduce to values before reductions in an abstraction body can take place. The answer to this question lies in the translation of the $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ construct, since it does not guard the processes in the image of the translation of M or N, allowing for reductions to take place in both. A careful look at the terms in the image $(\cdot)^*$ reveals that the β redex $(\lambda x.M)\,N$ translates (after some administrative reduction steps) to $\mathsf{let}\ !u = N^*\ \mathsf{in}\ M^*$, explaining the apparent discrepancy mentioned above.

Moreover, the logical basis of our approach provides the following property, relating type inhabitance in the λ-calculus and the session-typed π-calculus.

Theorem 2.19. *For any type T in the simply typed λ-calculus, T is inhabited iff $[T]$ and T^* are inhabited.*

Proof. The forward direction follows from Theorems 2.17 and 2.18, respectively, discarding the proof terms. The backward direction follows from the correctness of the logical translations, which are standard in the literature (the process to λ-term translations can be straightforwardly extracted from these proofs).

3 Soundness and Completeness

The correctness results we have presented are related to the well-formedness (through typing) of the terms in the image of each translation. We will now present a soundness

and completeness result for the dynamic behavior of the translations. Our translation from the linear λ-calculus to process calculus is looser than one might initially expect, that is, it allows for reductions that are not typical in the standard operational semantics of the linear λ-calculus. This means that w.r.t the standard operational semantics of the linear λ-calculus, our translation should be able to simulate all reductions, but the converse cannot possibly hold. We now make this remark precise by first stating the typical operational semantics for the linear λ-calculus and showing a completeness result for our translation (i.e., reductions in λ-terms can be matched by a small sequence of reductions in the processes in the image of the translation). We then extend the operational semantics with a rule that allows for reductions under the let-binder, showing the new operational semantics to be observationally equivalent to the standard reduction rules in a precise sense. With this extended rule, we show soundness of our translation with respect to the operational semantics.

Definition 3.1 (Linear λ-calculus Operational Semantics).

$$\frac{M \longrightarrow M'}{M\,N \longrightarrow M'\,N} \qquad \frac{}{(\lambda x.M)\,N \longrightarrow M\{N/x\}}$$

$$\frac{M \longrightarrow M'}{\mathsf{let}\ !u = M\ \mathsf{in}\ N \longrightarrow \mathsf{let}\ !u = M'\ \mathsf{in}\ N} \qquad \frac{}{\mathsf{let}\ !u = !M\ \mathsf{in}\ N \longrightarrow N\{M/u\}}$$

To accurately state the desired theorems, we need to characterize the relation between the substitutive behavior on the λ-calculus, where variables are substituted by λ-terms, and the π-calculus, where names are substituted by names. The idea is that instead of plugging in a term for a variable, we can think of names as "references" to a term (a channel name along which we communicate with a parallel process that encodes a given term). More precisely, we want to capture the "equivalence" of $[\![M\{N/x\}]\!]_z$ and $(\nu x)([\![M]\!]_z \mid [\![N]\!]_x)$, where x is a linear variable in M. For the unrestricted case, a similar equivalence is required, but using replicated processes. We combine these two key insights in the following relation.

Definition 3.2 (Substitution Lifting). *For a given linear λ-term M and process P, we say that P lifts the substitutions of M, written $M \triangleleft P$, if $P \equiv (\nu x_i, u_i, y_i)([\![R]\!]_z \mid [\![W_i]\!]_{x_i} \mid !u_i(x).[\![E_i]\!]_x)$ and $M = R\{W_i/x_i, E_i/y_i\}$.*

Intuitively, the relation above holds if P can be decomposed in such a way that we obtain the parallel composition of $[\![R]\!]_z$, which is the translation of a λ-term R on which no substitutions are actually carried out, and possibly many other processes that encode the λ-terms that are to be substituted into R (either linearly or in an unrestricted manner), to obtain the given term M. We now state the following completeness result for our translation of the linear λ-calculus.

Theorem 3.3 (Completeness of Translation). *If $\Gamma; \Delta \vdash M : A$ and $M \longrightarrow N$ then $[\![M]\!]_z \rightarrow^* P$ such that $N \triangleleft P$.*

As mentioned above, the processes in the image of our translation induce more non-determinism in the operational semantics of the linear λ-calculus. In particular, the process resulting from the translation of the term $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ allows for reductions to take place in both the translation of M and N, while the operational semantics

of Def. 3.1 do not. However, these extra reductions do not change the observable outcome of a λ-term, in a very precise sense. Consider the following operational semantics, consisting of the rules of of Def. 3.1 extended with an additional rule that allows for reductions under the let-binder:

Definition 3.4 (Extended Operational Semantics).

$$\frac{M \longrightarrow M'}{M \longmapsto M'} \qquad \frac{N \longmapsto N'}{\mathsf{let}\ !u = M\ \mathsf{in}\ N \longmapsto \mathsf{let}\ !u = M\ \mathsf{in}\ N'}$$

If we consider observables to be values (which is the standard observable in functional languages), the two operational semantics are equivalent in our typed setting. This can be made precise with the following theorem.

Theorem 3.5. *$M \longrightarrow^* V$ iff $M \longmapsto^* V$, where: $V ::= !M \mid \lambda x : T.M$*

Proof. The left to right direction is immediate. From right to left, the idea is that we can delay all the reductions of N in the additional rule until after M reduces to a term of the form $!M$ and the substitution into N takes place.

Theorem 3.6 (Soundness of Translation). *If $[\![M]\!]_z \rightarrow P$, with $\Gamma; \Delta \vdash M : A$ then there is Q and N such that $P \rightarrow^* Q$, $M \longmapsto N$ and $N \lhd Q$*

Given these two results, we can see that sharing and copying reduction arises from the typing discipline enforced by each translation, given the operational semantics for the linear λ-calculus considered above. This validates the translation from the linear λ-calculus to the π-calculus, and given that the translations from the λ to the linear λ-calculus have been shown to be sound and complete w.r.t the operational semantics in [15], these results extend to our full translations.

4 Concluding Remarks

There is a substantial body of work on the connection of functional and concurrent programming, both from the concurrency and proof theory communities. Milner [17] developed two embeddings of the λ-calculus in the π-calculus, corresponding to call-by-value and call-by-name. His focus was mainly operational, while ours is strictly driven by the logical considerations we have alluded to previously. Along these lines, Sangiorgi and Walker [19] provide an uniform presentation of encodings for call-by-value (in parallel or non parallel form), as well as call-by-need, but do not address sharing versus copying as we do here. Boudol's [5] work on the λ-calculus with multiplicities (arising from the study of Milner's call-by-name embedding), consists of a λ-calculus with inherently non-deterministic and parallel constructs for function application, drawing inspiration on many ideas from linear logic but not studying the deeper connections of the two systems. Yoshida et al. have studied sequentiality in the π-calculus [3] by introducing a typing discipline that allows for direct interpretations of call-by-value and call-by-name, presenting a full abstraction result for an encoding of PCF and connections to game semantics. This contrasts with our work which focuses on the concurrency

and parallelism that can be extracted from π-calculus encodings, instead of using the π-calculus as a tool for the study of sequential computation. Furthermore, their work does not explore connections with logic.

On the proof theory side, the connection of linear logic and concurrency has been explored in some detail, using proof net-like structures. Mazza's work on Multiport Interaction Nets [16], which are a non-deterministic extension of Interaction Nets [13], shows that these are a model of concurrent computation by encoding the full π-calculus. Faggian and Piccolo [7] perform a similar development for Ludics and the finitary linear π-calculus. A correspondence between polarised proof-nets and an IO-typed π-calculus is given in [11]. The line of work summarized above is quite different from ours, in that it is mostly concerned with semantic models for concurrency, derived from linear logic.

In work much closer to our own, Beffara [2] presents embeddings of the λ-calculus and a restricted version the $\lambda\mu$-calculus in a π-calculus derived from an interpretation of second-order classical linear logic. The λ-calculus translation is closely related to our copying translation. This work provides a family of call-by-value and by-name translations, in both classical and intuitionistic settings, and explores the duality of both reduction strategies by exploring the dualities of classical (linear) logic, while our work focuses on the issues of sharing and copying that can be derived from translations in the purely intuitionistic (and non-higher order) setting.

Finally, the work on Multilisp [10], which is an extension of Lisp with explicit parallelism, pioneered the concept of *futures*, which allow for the concurrent execution of function arguments and bodies. Our sharing reduction strategy captures this concept, by enabling such parallel execution, and thus validates the ideas of Multilisp in a typed setting, motivated by proof theory.

Conclusion and Future Work We have developed two logically motivated embeddings of the simply-typed λ-calculus in a session-typed π-calculus, using standard techniques for translating natural deduction into sequent calculus and a concurrent interpretation of the latter. The logical foundation provides a clean and canonical account of each step of our translation, relying only on well-established techniques from proof theory and their interpretation as functional programming. We have shown that the two translations we consider induce a copying and a sharing reduction strategy (i.e. parallel evaluation of shared sub-expressions – no implementation of the λ-calculus in the π-calculus in this sense has been proposed before), respectively, providing a logical understanding of these concurrent evaluation strategies. Finally, the typical call-by-value and call-by-need sequential reduction strategies can be understood as two extremal traces of the sharing reduction strategy, where in the former all reductions on arguments are done before those of the function body, and in the latter the evaluation of the argument is postponed until it is needed in the function body. For future work, we plan on investigating further the issues of observational equivalence at the process level and its impact on the translations, as well as a natural generalization to dependently typed languages, using dependent session types.

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, INTERFACES NGN-44 / 2009, and CITI.

References

1. A. Barber and G. Plotkin. Dual Intuitionistic Linear Logic. Technical Report LFCS-96-347, Univ. of Edinburgh, 1997.
2. E. Beffara. Functions as proofs as processes. *CoRR*, abs/1107.4160, 2011.
3. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the pi-calculus. In *TLCA'01*, pages 29–45, 2001.
4. M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.
5. G. Boudol. The lambda-calculus with multiplicities. Technical report, INRIA, 1993.
6. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, pages 222–236. Springer LNCS 6269, 2010.
7. C. Faggian and M. Piccolo. Ludics is a model for the finitary linear pi-calculus. In *TLCA 2007*, Lecture Notes in Computer Science, pages 148–162. Springer-Verlag, 2007.
8. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
9. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *POPL 1992, Proceedings the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 15–26, 1992.
10. R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.
11. K. Honda and O. Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comp. Sci.*, 411:2223–2238, 2010.
12. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, pages 122–138. Springer LNCS 1381, 1998.
13. Y. Lafont. Interaction nets. In *POPL'90*, pages 95–108, New York, NY, USA, 1990. ACM.
14. J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL 1990, Proceedings the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, 1990.
15. J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electr. Notes Theor. Comput. Sci.*, 1:370–392, 1995.
16. D. Mazza. Multiport interaction nets and concurrency. In M. Abadi and L. .de Alfaro, editors, *CONCUR'05*, volume 3653 of *Lecture Notes in Computer Sciences*, pages 21–35. Springer, 2005.
17. R. Milner. Functions as processes. *Math. Struct. in Comp. Sci.*, 2(2):119–141, 1992.
18. D. Sangiorgi. Pi-Calculus, Internal Mobility, and Agent Passing Calculi. *Theoretical Computer Science*, 167(1&2):235–274, 1996.
19. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
20. B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11*, pages 161–172, New York, NY, USA, 2011. ACM.
21. S. van Bakel and M. G. Vigliotti. A logical interpretation of the λ-calculus into the π-calculus, preserving spine reduction and types. In *CONCUR'09*, pages 84–98, Bologna, Italy, Sept. 2009. Springer LNCS 5710.
22. P. Wadler. A syntax for linear logic. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *MFPS'93*, pages 513–529, New Orleans, Louisiana, Apr. 1993. Springer-Verlag LNCS 647.
23. C. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.