A logical interpretation of the λ-calculus into the π-calculus, preserving spine reduction and types

Steffen van Bakel and Maria Grazia Vigliotti

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK
svb@doc.ic.ac.uk, mgv98@doc.ic.ac.uk

Abstract. We define a new, output-based encoding of the λ-calculus into the asynchronous π-calculus – enriched with pairing – that has its origin in mathematical logic, and show that this encoding respects one-step spine-reduction up to substitution, and that normal substitution is respected up to similarity. We will also show that it fully encodes lazy reduction of closed terms, in that term-substitution as well as each reduction step are modelled up to similarity. We then define a notion of type assignment for the π-calculus that uses the type constructor $\rightarrow$, and show that all Curry-assignable types are preserved by the encoding.

Introduction

In this paper we present a new encoding of terms of Church's λ-calculus [11, 7] into Milner's π-calculus [18] that respects reduction, and define a new notion of type assignment for π so that processes will become witnesses for the provable formulae.

Sangiorgi [21] states good reasons for obtaining an expressive encoding:

- From the process calculi point of view, to gain deeper insight into its theory.
- From the λ-calculus point of view, to provide the opportunity to study λ-terms in different contexts than the sequential one.
- The λ-calculus is a model for functional language programming; these languages have never been very efficient, and one way of improving efficiency is to use parallel implementation.

So therefore, it is important to understand which relation is held between the two paradigms. Research in the direction of encodings of λ-terms was started by Milner in [18]; he defined an input-based encoding, and showed that the interpretation of closed λ-terms respects *lazy* reduction up to substitution. Milner also defined another input-based encoding that respects *call-by-value* reduction up to substitution, but the latter had fewer followers. An input-based interpretation of the λ-calculus into the π-calculus has also been studied by Sangiorgi [23], but in the context of the higher-order π-calculus, by Honda *et al.* [17] with a rich type theory, and by Thielecke [25] in the context of continuation passing style programming languages.

For many years, it seemed that the first and final word on the encoding of the λ-calculus has been said by Milner; in fact, Milner's encoding has set a milestone in the comparison of the two paradigms, and all the above mentioned systems present variants of Milner's encoding. Sangiorgi [20] says:

> " *It seems established that* [Milner's encoding] *is canonical, i.e. it is the 'best' or 'simplest' encoding of the lazy λ-calculus into π-calculus. Nevertheless, one has to think carefully about it – in particular at the encoding of application – to understand that it really does work.* "

We present in this paper a *conceptually different* encoding, that not only respects lazy reduction, but also the (larger) notion of spine reduction, and is easier to understand. Essentially following the structure of Milner's proof, central to our approach is the interpretation of the *explicit substitution* version of spine reduction, which allows us to establish a clear connection between term-substitution in the λ-calculus, and the simulation of this operation in the π-calculus via channel-name passing.

We also investigate our interpretation in relation to the type system presented in [4]. This system provides a logical view to the π-calculus, where π-processes can be witnesses of formulae that are provable (within the implicative fragment) in classical logic, as was shown in [4]. That system is different from standard type systems for π as it does not contain any channel information, and in that it expresses implication. We show that our encoding preserves types assignable to λ-terms in Curry's system. Through this type preservation result we show that our encoding also respects the *functional* behaviour of terms, as expressed via assignable types. In this way we establish a deeper relationship between sequential/applicative and concurrent paradigms.

The results on the type system that we present here determines the choice of π-calculus used for the encoding: we use the asynchronous π-calculus enriched with *pairs* of names [1]. In principle, our encoding could be adapted to the synchronous monadic π-calculus, however we would not be able to achieve the preservation of assignable types. Our encoding takes inspiration from, but it is a much improved version of, the encoding of λ-terms in to the sequent calculus $\mathcal{X}$ [5, 6] – a first variant was defined by Urban [26, 27]; $\mathcal{X}$ is a sequent calculus that enjoys the Curry-Howard correspondence for Gentzen's LK [13] – and the encoding of $\mathcal{X}$ into π-calculus as defined in [4].

Our work not only sheds new light on the connection between sequential and concurrent computation but also established a firm link between logic and process calculi. The relation between process calculi and classical logic as first reported on in [4] is an interesting and very promising area of research (similar attempts we made in the context of natural deduction [17], and linear logic [9]). A preliminary interpretation of λ-terms in to the π-calculus was shown in [4]; in this paper we improve the interpretation and strengthen operational correspondence results, by establishing the relationship between the π-calculus ad explicit substitution, and by considering spine reduction.

In summary, the main achievements of this paper are:

- an output-based encoding of the λ-calculus into the asynchronous π-calculus with pairing is defined that preserves spine reduction with explicit substitution for all terms up to contextual equivalence, and, by inclusion, for lazy reduction with explicit substitution;
- our encoding also respects implicit substitution, and respects lazy reduction for closed terms up to simulation;
- our encoding preserves assignable Curry types for λ-terms, with respect to the context assignment system for π from [4].

Paper outline. In Sec. 1, we repeat the definition of the asynchronous π-calculus with pairing, and in Sec. 2 that of the λ-calculus, where we present the notion of explicit spine reduction '$\to_{\mathbf{xS}}$' which takes a central role in this paper; in Sec. 3 we also briefly discuss Milner's interpretation result for the lazy λ-calculus. Then, in Sec. 4, we will define an encoding where terms are interpreted under *output* rather than *input* (as in Milner's), and show that $\to_{\mathbf{xS}}$ is respected by our interpretation; we will also show a simulation result for full β-reduction. Finally, in Sec. 5, we give a notion of (type) context assignment on processes in π, and show that our interpretation preserves types. In fact, this result is the main motivation for our interpretation, which is therefore *logical.*

1 The asynchronous π-calculus with pairing

The notion of asynchronous π-calculus that we consider in this paper is the one used also in [1, 4], and is different from other systems studied in the literature [15] in a number of aspects: we add pairing, and introduce the *let*-construct to deal with inputs of pairs of names that get distributed. The main reason for the addition of pairing [1] lies in the fact that we want to preserve implicate type assignment. The π-calculus is an input-output calculus, where terms have not just more than one input, but also more than one output. This is similar to what we find in Gentzen's LK, where right-introduction of the arrow is represented by

$$(\Rightarrow R): \quad \frac{\Gamma \vdash_{\mathrm{LK}} \Delta}{\Gamma' \vdash_{\mathrm{LK}} A \Rightarrow B, \Delta'} \, (\Gamma = \Gamma', A \;\&\; \Delta = B, \Delta')$$

where Γ and Δ are multi-sets of formulae. Notice that only *one* of the possible formulae is selected from the right context, and *two* formulae are selected in *one* step; when searching for a Curry-Howard correspondence, this will have to be reflected in the (syntactic) witness of the proof. So if we want to model this in π, i.e. want to express function construction (abstraction), we need to bind *two* free names, one as name for the input of the function, and the other as name for its output. We can express that a process P acts as a function *only* when fixing (binding) *both* an input *and* an output *simultaneously*, i.e. in *one* step; we use pairing exactly for this: interfaces for functions are modelled by sending and receiving *pairs* of names.

Below, we will use '$\circ$' for the generic variable, and introduce a structure over names, such that not only names but also pairs of names can be sent (but not a pair of pairs); this way a channel may pass along either a name or a pair of names.

Definition 1. *Channel names* and *data* are defined by:

$$a, b, c, d, x, y, z \quad \textit{names} \qquad\qquad p ::= a \mid \langle a, b \rangle \quad \textit{data}$$

Notice that pairing is *not* recursive. Processes are defined by:

$P, Q ::= 0$	*Nil*	$\mid a(x).P$	*Input*
$\mid P \mid Q$	*Composition*	$\mid \overline{a}\langle p \rangle$	*(Asynchronous) Output*
$\mid !P$	*Replication*	$\mid \mathit{let}\, \langle x, y \rangle = z \;\mathit{in}\; P$	*Let construct*
$\mid (\nu a)\, P$	*Restriction*		

We abbreviate $a(x).\mathit{let}\,\langle y, z\rangle = x \;\mathit{in}\; P$ by $a(\langle y, z\rangle).P$, and $(\nu m)\,(\nu n)\,P$ by $(\nu mn)\,P$.

A (process) context is simply a term with a hole $[\cdot]$.

Definition 2 (Congruence). The structural congruence is the smallest equivalence relation closed under contexts defined by the following rules:

$$\begin{array}{rlrl} P \mid \mathbf{0} & \equiv P & (\nu m)\,(\nu n)\,P & \equiv (\nu n)\,(\nu m)\,P \\ P \mid Q & \equiv Q \mid P & (\nu n)\,(P \mid Q) & \equiv P \mid (\nu n)\,Q \quad \textit{if } n \notin \mathit{fn}(P) \\ (P \mid Q) \mid R & \equiv P \mid (Q \mid R) & !\,P & \equiv P \mid !\,P \\ (\nu n)\,\mathbf{0} & \equiv \mathbf{0} & \textit{let}\,\langle x,y\rangle = \langle a,b\rangle\,\textit{in}\,R & \equiv R[a/x,b/y] \end{array}$$

Definition 3. The *reduction relation* over the processes of the π-calculus is defined by following (elementary) rules:

$$\begin{array}{rl} (\textit{synchronisation}): & \overline{a}\langle b\rangle \mid a(x).\,Q \;\rightarrow_\pi\; Q[b/x] \\ (\textit{hiding}): & P \rightarrow_\pi P' \;\Rightarrow\; (\nu n)\,P \rightarrow_\pi (\nu n)\,P' \\ (\textit{composition}): & P \rightarrow_\pi P' \;\Rightarrow\; P \mid Q \rightarrow_\pi P' \mid Q \\ (\textit{congruence}): & P \equiv Q \;\&\; Q \rightarrow_\pi Q' \;\&\; Q' \equiv P' \;\Rightarrow\; P \rightarrow_\pi P' \end{array}$$

We write $\rightarrow_\pi^*$ for the reflexive and transitive closure of this relation.

Notice that $\overline{a}\langle\langle b,c\rangle\rangle \mid a(\langle x,y\rangle).\,Q \;\rightarrow_\pi\; Q[b/x,c/y]$.

Definition 4. 1. We write $P \downarrow n$ (P *outputs on* n) if $P \equiv (\nu b_1 \ldots b_m)\,(\overline{n}\langle p\rangle \mid Q)$ for some Q, where $n \neq b_1 \ldots b_m$.
2. We write $P \Downarrow n$ (P *will output on* n) if there exists Q such that $P \rightarrow_\pi^* Q$ and $Q \downarrow n$.
3. We write $P \sim_{\mathsf{C}} Q$ (and call P and Q *contextually equivalent*) if, for all contexts $\mathsf{C}[\cdot]$, and for all n, $\mathsf{C}[P] \Downarrow n \iff \mathsf{C}[Q] \Downarrow n$.

Definition 5 ([16]). *Barbed contextual simularity* is the largest relation $\overset{\pi}{\preceq}$ such that $P \overset{\pi}{\preceq} Q$ implies:

- for each name n, if $P \downarrow n$ then $Q \Downarrow n$;
- for any context $\mathsf{C}[\cdot]$, if $\mathsf{C}[P] \rightarrow_\pi P'$, then for some Q', $\mathsf{C}[Q] \rightarrow_\pi^* Q'$ and $P' \overset{\pi}{\preceq} Q'$.

2 The Lambda Calculus (and variants thereof)

We assume the reader to be familiar with the λ-calculus; we just repeat the definition of the relevant notions.

Definition 6 (Lambda terms and β-contraction [7]).

1. The set Λ of λ-*terms* is defined by the grammar:
$$M, N ::= x \mid \lambda x.M \mid MN$$
2. The reduction relation $\rightarrow_\beta$ is defined by the rules:
$$(\lambda x.M)N \;\rightarrow\; M[N/x] \qquad M \rightarrow N \;\Rightarrow\; \begin{cases} ML \rightarrow NL \\ LM \rightarrow LN \\ \lambda x.M \rightarrow \lambda x.N \end{cases}$$
3. *Lazy*[1] reduction [2] $\rightarrow_{\mathrm{L}}$ is defined by limiting the reduction relation to:
$$(\lambda x.M)N \;\rightarrow\; M[N/x] \qquad M \rightarrow N \;\Rightarrow\; ML \rightarrow NL$$

[1] This reduction relation is sometimes also known as 'Call-by-Name'; since this is an overloaded concept, we stick to the terminology 'lazy'; the definition here is the one used in [18].

4. We define *spine* reduction[2] $\to_{\mathrm{S}}$ by limiting reduction to:

$$(\lambda x.M)N \to M[N/x] \qquad M \to N \;\Rightarrow\; \begin{cases} ML \to NL \\ \lambda x.M \to \lambda x.N \end{cases}$$

Notice that spine reduction is aptly named, since all reductions take place on the spine of the λ-tree (see [8]): searching for a redex, starting from the root, we can walk 'down' and turn 'left', but not turn 'right', so stay on the spine of the tree. This notion of reduction is shown to be head-normalising in [8]; in fact, the normal forms for spine reduction are exactly the head-normal forms for normal reduction [28].

Example 7. Spine reduction encompasses lazy reduction:

$$(\lambda x.(\lambda y.M)N)L \to_{\mathrm{S}} \begin{cases} (\lambda x.M[N/y])L \\ ((\lambda y.M)N)[L/x] \end{cases}$$

whereas only $(\lambda x.(\lambda y.M)N)L \to_{\mathrm{L}} ((\lambda y.M)N)[L/x]$.

In view of the importance of substitution also in Milner's result (see Thm. 14), rather than directly interpreting the spine calculus λS, in this paper we will treat $\lambda\mathbf{x}$S, a version with explicit substitution, *à la* Bloo and Rose's calculus $\lambda\mathbf{x}$ [10].

Definition 8 ($\lambda\mathbf{x}$S). The syntax of $\lambda\mathbf{x}$S is an extension of that of the λ-calculus:

$$M, N ::= x \mid \lambda x.M \mid MN \mid M\langle x = N\rangle$$

The explicit variant $\to_{\mathbf{x}\mathrm{S}}$ of spine reduction is now defined as follows. We take the basic rules:

$$(\lambda x.M)N \to M\langle x = N\rangle \qquad M \to N \;\Rightarrow\; \begin{cases} ML \to NL \\ \lambda x.M \to \lambda x.N \end{cases}$$

(notice the use of $\langle x = N\rangle$ rather than $[N/x]$ in the first rule). The 'propagation rules' for substitution are defined as:

$$M\langle x = N\rangle \to M \; (x \notin fv(M)) \qquad (\lambda y.M)\langle x = N\rangle \to \lambda y.(M\langle x = N\rangle)$$
$$(xM_1 \cdots M_n)\langle x = N\rangle \to (NM_1 \cdots M_n)\langle x = N\rangle$$

Remark 9. We deviate above from the original definition of reduction $\to_{\mathbf{x}}$ in [10], which included the rules:

$$(ML)\langle x = N\rangle \to M\langle x = N\rangle L\langle x = N\rangle \qquad x\langle x = N\rangle \to N$$
$$(\lambda y.M)\langle x = N\rangle \to \lambda y.(M\langle x = N\rangle) \qquad y\langle x = N\rangle \to y$$

Since spine reduction focusses on the head of a term, we postpone the substitution on other parts, and only 'work' on the head.[3]

It is easy to show that $M \to_{\mathbf{x}\mathrm{S}} N$ implies $M \to_{\mathbf{x}} N$, and that if $M \to_{\mathbf{x}\mathrm{S}} N$, then there exists a pure term L (not containing $\langle \cdot = \cdot \rangle$) such that $N \to_{\mathbf{x}} L$, and in this reduction only the substitution rules of $\lambda\mathbf{x}$ are applied. Since spine reduction reduces a term M to head-normal form, if it exists, this implies that also $\to_{\mathbf{x}\mathrm{S}}$ reduces to head-normal form, albeit with perhaps some substitutions still pending.

[2] In [14], essentially following [8], spine reduction is defined by "*just contracting redexes that are on the spine*"; *head* spine reduction is mentioned, but not defined, in [24].

[3] This appears to be the implicit approach of [18] (see Lem. 4.5, case 3).

Example 10. Some terms leave substitutions after reducing: $(\lambda z.yz)N \to_{\mathbf{xs}} yz\langle z=N\rangle$.

We can reduce $(\lambda x.(\lambda z.(\lambda y.M)x))N$ in two different ways:

$$\begin{array}{lll} (\lambda x.(\lambda z.(\lambda y.M)x))N & \to_{\mathbf{xs}} & \\ (\lambda z.(\lambda y.M)x)\langle x=N\rangle & \to_{\mathbf{xs}} & \\ \lambda z.((\lambda y.M)x)\langle x=N\rangle & \to_{\mathbf{xs}} & \\ \lambda z.(M\langle y=x\rangle\langle x=N\rangle) & & \end{array} \qquad \begin{array}{lll} (\lambda x.(\lambda z.(\lambda y.M)x))N & \to_{\mathbf{xs}} & \\ (\lambda x.(\lambda z.(M\langle y=x\rangle)))N & \to_{\mathbf{xs}} & \\ \lambda z.(M\langle y=x\rangle)\langle x=N\rangle & \to_{\mathbf{xs}} & \\ \lambda z.(M\langle y=x\rangle\langle x=N\rangle) & & \end{array}$$

3 Milner's input-based lazy encoding

Milner defines an encoding of the λ-calculus into the (synchronous, monadic) π-calculus [18][4], and shows some correctness results. This encoding is inspired by the normal semantics of λ-terms, which states for abstraction:

$$[\![\lambda x.M]\!]^{\mathcal{M}}_{\xi} = G(\lambda d \in \mathcal{M}.[\![M]\!]^{\mathcal{M}}_{\xi(d/x)})$$

Here the body of the abstraction is interpreted in the updated valuation, where now also x is mapped to d, an arbitrary element of the domain. So, also in the encoding, instead of executing $M[N/x]$, M is executed in an environment that binds N to the variable x; this leads to:

Definition 11 (Milner's interpretation [18]). Input-based encoding of λ-terms into the π-calculus is defined by:

$$\begin{array}{rcll} [\![x]\!]\,a & \triangleq & \overline{x}\langle a\rangle & x \neq a \\ [\![\lambda x.M]\!]\,a & \triangleq & a(x).a(b).[\![M]\!]\,b & b \textit{ fresh} \\ [\![MN]\!]\,a & \triangleq & (\nu c)\,([\![M]\!]\,c \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := N]\!])) & c,z \textit{ fresh} \\ [\![x := M]\!] & \triangleq & !\,x(w).[\![M]\!]\,w & w \textit{ fresh} \end{array}$$

(Milner calls $[\![x := M]\!]$ an "*environment entry*"; it could be omitted from the definition above, but is of use separately.) Here a is the link along which $[\![M]\!]$ receives its argument; this is used to communicate with the interpretation of the argument, as made clear in the third case, were the input channel of the left-hand term is used to send the name over on which the right-hand term will receive its input.

Milner's initial approach has since become standard, and is also used in [20–22, 17]; in fact, as mentioned in the introduction, Sangiorgi considers it canonical [20].

Notice that both the body of the abstraction (M) and the argument in an application (N) get positioned under an input, and that therefore reductions inside these subterms cannot be modelled, and the simulation via the encoding is limited to lazy reduction[5].

Example 12. Using $[\![\cdot]\!]\,\cdot$, the interpretation of a β-redex (only) reduces as follows:

$$\begin{array}{lll} [\![(\lambda x.M)N]\!]\,a & \triangleq & \\ (\nu c)\,(c(x).c(b).[\![M]\!]\,b \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := N]\!])) & \to_{\pi}^{*} & \\ (\nu z)\,([\![M[z/x]]\!]\,a \mid [\![z := N]\!]) & = & (z \notin [\![M]\!]\,b) \\ (\nu x)\,([\![M]\!]\,a \mid [\![x := N]\!]) & \triangleq & (\nu x)\,([\![M]\!]\,a \mid !\,x(w).[\![N]\!]\,w) \end{array}$$

[4] [18] also deals with Call-By-Value, which we will not consider here.

[5] It is possible to improve on this result by extending the notion of reduction or congruence on π, by adding, for example, $P \to_{\pi} Q \Rightarrow x(v).P \to_{\pi} x(v).Q$, but that is not our intention here; we aim just to compare our result with Milner's.

Now reduction can continue in (the interpretation of) M, but not in N that is still guarded by the input on x, which will not be used until the evaluation of $[\![M]\!]\,a$ reaches the point where output is generated over x.

In fact, Milner shows that all interpretations of closed λ-terms reduce to terms of this shape (Thm. 14). Notice that the result $(\nu x)\,([\![M]\!]\,a \mid [\![x := N]\!])$ is not the same as $[\![M[N/x]]\!]\,a$, and also does not reduce to that term, as illustrated by the following:

Example 13.
$$
\begin{array}{rl}
[\![(\lambda x.xx)(\lambda y.y)]\!]\,a & \triangleq \\
\quad (\nu c)\,(c(x).c(b).[\![xx]\!]\,b \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := \lambda y.y]\!])) & \to_\pi^* \\
\quad (\nu z)\,([\![zz]\!]\,a \mid [\![z := \lambda y.y]\!]) & \triangleq \\
\quad (\nu z)\,((\nu c)\,(\overline{z}\langle c\rangle \mid (\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!])) \mid [\![z := \lambda y.y]\!]) & \equiv \\
\quad (\nu zc)\,(\overline{z}\langle c\rangle \mid (\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!]) & \\
\qquad\qquad \mid z(w).[\![\lambda y.y]\!]\,w \mid [\![z := \lambda y.y]\!]) & \to_\pi \\
\quad (\nu zc)\,((\nu z_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!]) \mid [\![\lambda y.y]\!]\,c \mid [\![z := \lambda y.y]\!]) & \equiv \\
\quad (\nu zcz_1)\,(\overline{c}\langle z_1\rangle.\overline{c}\langle a\rangle.[\![z_1 := z]\!] \mid c(y).c(b).\overline{y}\langle b\rangle \mid [\![z := \lambda y.y]\!]) & \to_\pi^* \\
\quad (\nu zz_1)\,([\![z_1 := z]\!] \mid \overline{z_1}\langle a\rangle \mid [\![z := \lambda y.y]\!]) & \to_\pi \\
\quad (\nu zz_1)\,(\overline{z}\langle a\rangle \mid [\![z_1 := z]\!] \mid [\![z := \lambda y.y]\!]) & \equiv \\
\quad (\nu zc)\,(\overline{z}\langle a\rangle \mid z(w).[\![\lambda y.y]\!]\,w \mid [\![z := \lambda y.y]\!]) & \to_\pi \\
\quad (\nu z)\,([\![\lambda y.y]\!]\,a \mid [\![z := \lambda y.y]\!]) & \equiv [\![\lambda y.y]\!]\,a
\end{array}
$$

Notice that we executed the only possible communications, and that in the reduction path no term corresponds to $(\nu c)\,([\![\lambda y.y]\!]\,c \mid (\nu z)\,(\overline{c}\langle z\rangle.\overline{c}\langle a\rangle.[\![z := \lambda y.y]\!]))$ (i.e. $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$). Of course reducing the term $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$ will also yield $[\![\lambda y.y]\!]\,a$, but we can only show that $[\![(\lambda x.xx)(\lambda y.y)]\!]\,a$ and $[\![(\lambda y.y)(\lambda y.y)]\!]\,a$ have a *common reduct*, not that the first reduces to the second.

Milner states the correctness for his interpretation with respect to lazy reduction as:

Theorem 14 ([18]). *For all closed λ-terms M, either:*

1. $M \to_{\mathrm{L}} \lambda y.R[\overrightarrow{N/x}]$, *and* $[\![M]\!]\,u \to_\pi (\overrightarrow{\nu x})\,([\![\lambda y.R]\!]\,u \mid [\![\overrightarrow{x := N}]\!])$, *or*
2. *both M and $[\![M]\!]\,u$ diverge.*

It is worthwhile to note that, although not mentioned in [18], in the proof of this result Milner treats the substitution as *explicit*, not as *implicit*. In fact, although stated with implicit substitution, Milner's result does not show that lazy reduction (with implicit substitution) is fully modelled, but only 'up to substitution'; as shown in Ex. 13, it is impossible to reduce the encoding of $(\lambda x.xx)(\lambda y.y)$ to that of $(\lambda y.y)(\lambda y.y)$.

We quickly found that we (also) could not model implicit substitution, and reverted to *explicit* substitution; however, $(\lambda x.xx)(\lambda y.y) \to_{\mathbf{xs}} ((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$, and we *can* show that the encoding of $(\lambda x.xx)(\lambda y.y)$ runs to that of $((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$, as shown in Ex. 23.

4 A logical, output-based encoding of λ-terms

In this section, we will show that it is possible to deviate from Milner's original encoding, and actually making a gain in the process. Inspired by the relation between natural deduction and the sequent calculus [13], interpreting terms under *output* rather than under *input*, and using the π-calculus with pairing, we can define a different encoding of the λ-calculus into the π-calculus that preserves not just lazy reduction, but also the larger notion of spine reduction.

Our encoding follows from – but is an improvement of – the concatenation of the encoding of the λ-calculus into $\mathcal{X}$ (which established a link between natural deduction and the sequent calculus), defined in [6], and the interpretation of $\mathcal{X}$ into the π-calculus as defined in [4]. The main objective of our encoding is to show the preservation of type assignment; pairing is used in order to be able to effectively represent arrow types.

The idea behind our encoding originates from the observation that, in the lambda calculus, all input is named, but output is anonymous. Input (i.e. a *variable*) is named to serve as a destination for the substitution; output need not be named, since all terms have only one result (represented by the term itself), which is used *in sito*[6]. Translating into the (multi-output) π-calculus, this locality property no longer holds; we need to specify the destination of a term, by naming its output: this is what the encoding does.

We explicitly convert '*an output sent on a is to be received as input on b*' via '$a(\circ).\overline{b}\langle\circ\rangle$' (call a *forwarder* in [16]), which for convenience is abbreviated into $a{=}b$.

Definition 15 (Output-based interpretation). The mapping $[\![\cdot]\!]\cdot$ is defined by[7]:

$$\begin{array}{rcll} [\![x]\!]a & \triangleq & x(\circ).\overline{a}\langle\circ\rangle & x \neq a \\ [\![\lambda x.M]\!]a & \triangleq & (\nu xb)\,([\![M]\!]b \mid \overline{a}\langle\langle x,b\rangle\rangle) & b \textit{ fresh} \\ [\![MN]\!]a & \triangleq & (\nu c)\,([\![M]\!]c \mid c(\langle b,d\rangle).\,(!\,[\![N]\!]b \mid d{=}a)) & b,c,d \textit{ fresh} \end{array}$$

In particular:
- we see a variable x as an input channel, and we need to retransmit its input to the output channel a that we interpret it under;
- for an abstraction $\lambda x.M$, we give the name b to the output of M; that M has input x and output b gets sent out over a, which is the name of $\lambda x.M$, so that a process that wants to call on this functionality, knows which channel to send the input to, and on which channel to pick up the result[8];
- for an application MN, the output of M, transmitted over c, is received as a pair $\langle b,d\rangle$ of input-output names in the right-hand side; the received input b name is used as output name for N, enabling the simulation of substitution, and the received output name d gets redirected to the output of the application a.

[6] In terms of *continuations*, the continuation of a term is not mentioned, since it is the current.

[7] We could have defined our encoding directly in the standard π-calculus:

$$\begin{array}{rcl} [\![x]\!]'a & \triangleq & !\,x(\circ).\overline{a}\langle\circ\rangle \\ [\![\lambda x.M]\!]'a & \triangleq & (\nu xb)\,([\![M]\!]'b \mid \overline{a}\langle x\rangle.\overline{a}\langle b\rangle) \\ [\![MN]\!]'a & \triangleq & (\nu c)\,([\![M]\!]'c \mid c(b).c(d).\,(!\,[\![N]\!]'b \mid d{=}a)) \end{array}$$

without losing any of the reduction results for our encoding, but this has additional replication, and is less suited for type assignment (see Sect. 5).

[8] This view of computation is exactly that of the calculus $\mathcal{X}$.

Notice that only one replication is used, on the argument in an application; this corresponds, as usual, to the implementation of the (distributive) substitution on λ-terms. Also, every $[\![N]\!]a$ is a process that outputs on a non-hidden name a (albeit perhaps not actively, as in the third case, where it will not be activated until input is received on the channel c, in which case it is used to output the data received in on the channel d that is passed as a parameter), and that this output is unique, in the sense that a is the only output channel, is only used once, and for output only. The structure of the encoding of application corresponds, in fact, to how Gentzen encodes *modus ponens* in the sequent calculus [13]: see [6], Thm. 4.8, and the proof of Thm. 31 below.

Example 16. 1. $[\![(\lambda y.P)Q]\!]a \quad \stackrel{\Delta}{=}$
$(\nu c)\,((\nu y b_1)\,([\![P]\!]b_1 \mid \overline{c}\langle\langle y, b_1\rangle\rangle) \mid c(\langle b, d\rangle).\,(!\,[\![Q]\!]b \mid d{=}a)) \;\rightarrow_\pi$
$(\nu y b)\,([\![P]\!]b \mid !\,[\![Q]\!]y \mid b{=}a)$

In short, the encoding of the redex $(\lambda y.P)Q$ will yield a communication that receives on the input channel called y in the interpretation of P, and the interpretation of Q being output on y.

2. $[\![\lambda x.x]\!]a \stackrel{\Delta}{=} (\nu x b)\,(x(\circ).\overline{b}\langle\circ\rangle \mid \overline{a}\langle\langle x, b\rangle\rangle)$.
3. $[\![xN]\!]a = (\nu c)\,([\![x]\!]c \mid c(\langle b, d\rangle).\,(!\,[\![N]\!]b \mid d{=}a)) \succeq^{\pi} x(\langle b, d\rangle).\,(!\,[\![N]\!]b \mid d{=}a)$
This term correctly expresses that computation is halted until on x we send the input-output interface of a function, which will then communicate with the output channel of N as its input channel.

Notice that the second term cannot input; it is easy to check that this is true for the the interpretation of all closed λ-terms, since we can show the following:

Property 17. $fn([\![M]\!]a) = fv(M) \cup \{a\}$.

The following result, which states that we can safely rename the (hidden) output of an encoded λ-term, is needed below:

Lemma 18. $(\nu a)\,(a{=}e \mid [\![N]\!]a) \sim_{\mathsf{c}} [\![N]\!]e$.

Using this result, we can show that

$$(\nu x b)\,([\![M]\!]b \mid !\,[\![N]\!]x \mid b{=}a) \sim_{\mathsf{c}} (\nu x)\,([\![M]\!]a \mid !\,[\![N]\!]x)$$

Following on from Ex. 16, we can now define

Definition 19. We extend our interpretation to $\lambda\mathbf{x}\mathrm{s}$, adding

$$[\![M\langle x = N\rangle]\!]a = (\nu x)\,([\![M]\!]a \mid !\,[\![N]\!]x)$$

As in [18, 21, 23], we can show a reduction preservation result; however, not by just restricting to (explicit) lazy reduction, but to the larger system for explicit spine reduction for the λ-calculus. Notice that, essentially following Milner, by using the reduction relation $\rightarrow_{\mathbf{x}\mathrm{s}}$, we show that our interpretation respects reduction upto substitution; however, we do not require the terms to be closed:

Theorem 20 ($[\![\cdot]\!]\cdot$ preserves $\rightarrow_{\mathbf{x}\mathrm{s}}$). *If $M \rightarrow_{\mathbf{x}\mathrm{s}} N$, then $[\![M]\!]a \sim_{\mathsf{c}} [\![N]\!]a$.*

So, perhaps contrary to expectation, since abstraction is not encoded using *input*, we can without problem model a reduction under a λ-abstraction. Moreover, the only extra property we use is the renaming of the output under which λ-terms are encoded (Lem. 18).

Let us illustrate this result via a concrete example.

Example 21.
$$
\begin{array}{lll}
[\![(\lambda x.(\lambda z.(\lambda y.M)x))N]\!]a & \stackrel{\Delta}{=}, \equiv & \\
(\nu c x b_1)\,([\![(\lambda z.(\lambda y.M)x)]\!]b_1 \mid \overline{c}\langle\langle x, b_1\rangle\rangle \mid c(\langle b, d\rangle).\,(!\,[\![N]\!]b \mid d{=}a)) & \to_\pi & \\
(\nu x b_1)\,([\![(\lambda z.(\lambda y.M)x)]\!]b_1 \mid !\,[\![N]\!]x \mid b_1{=}a) & \stackrel{\Delta}{=}, \equiv & \\
(\nu x b_1 z b_2 c_1 y b_3)\,([\![M]\!]b_3 \mid \overline{c_1}\langle\langle y, b_3\rangle\rangle \mid & & \\
\qquad c_1(\langle b, d\rangle).\,(!\,[\![x]\!]b \mid d{=}b_2) \mid \overline{b_1}\langle\langle z, b_2\rangle\rangle \mid !\,[\![N]\!]x \mid b_1{=}a) & \to_\pi & (c_1) \\
(\nu x b_1 z b_2 y b_3)\,([\![M]\!]b_3 \mid !\,[\![x]\!]y \mid b_3{=}b_2 \mid \overline{b_1}\langle\langle z, b_2\rangle\rangle \mid !\,[\![N]\!]x \mid b_1{=}a) & \to_\pi & (b_1) \\
(\nu x z b_2 y b_3)\,([\![M]\!]b_3 \mid !\,[\![x]\!]y \mid b_3{=}b_2 \mid \overline{a}\langle\langle z, b_2\rangle\rangle \mid !\,[\![N]\!]x) & \sim_{\mathsf{c}} & (18) \\
(\nu x z b_2 y)\,([\![M]\!]b_2 \mid !\,[\![x]\!]y \mid \overline{a}\langle\langle z, b_2\rangle\rangle \mid !\,[\![N]\!]x) & \equiv & \\
(\nu z b)\,((\nu x)\,((\nu y)\,([\![M]\!]b \mid !\,[\![x]\!]y) \mid !\,[\![N]\!]x) \mid \overline{a}\langle\langle z, b\rangle\rangle) & \stackrel{\Delta}{=} & \\
[\![\lambda z.M\langle y{=}x\rangle\langle x{=}N\rangle]\!]a & &
\end{array}
$$

In the proof of Thm. 20, in only two places do we perform a renaming via Lem. 18, so use that $(\nu a)\,(a{=}e \mid [\![N]\!]a) \sim_{\mathsf{c}} [\![N]\!]e$. In both cases, the term we rename occurs at the head, and – assuming the reduction terminates – either N reduces to an abstraction $\lambda z.N'$, and then (wlog)

$$
\begin{array}{ll}
(\nu a)\,(a{=}e \mid [\![N]\!]a) & \to_\pi \\
(\nu a)\,(a{=}e \mid (\nu z b)\,([\![N']\!]b \mid \overline{a}\langle\langle z, b\rangle\rangle)) & \equiv \\
(\nu a z b)\,(a{=}e \mid [\![N']\!]b \mid \overline{a}\langle\langle z, b\rangle\rangle) & \to_\pi \\
(\nu z b)\,([\![N']\!]b \mid \overline{e}\langle\langle z, b\rangle\rangle) &
\end{array}
$$

or N reduces to a variable, and the renaming need not be executed. So when performing a reduction on $[\![M]\!]a$, where M has a normal form with respect to $\to_{\mathbf{xs}}$ (a head-normal form with respect to $\to_\beta$), these renamings can be postponed.

Corollary 22. If M has a normal form N (wrt $\to_{\mathbf{xs}}$), then $[\![M]\!]a \to_\pi [\![N]\!]a$.

Example 23. By Thm.20, $[\![(\lambda x.xx)(\lambda y.y)]\!]a \sim_{\mathsf{c}} (\nu x)\,([\![xx]\!]a \mid !\,[\![\lambda y.y]\!]x)$; but we *can* run the π-process without using the equivalence relation:

$$
\begin{array}{lll}
[\![(\lambda x.xx)(\lambda y.y)]\!]a & \stackrel{\Delta}{=}, \equiv & \\
(\nu c x b_1)\,([\![xx]\!]b_1 \mid \overline{c}\langle\langle x, b_1\rangle\rangle \mid c(\langle b, d\rangle).\,(!\,[\![\lambda y.y]\!]b \mid d{=}a)) & \to_\pi & \\
(\nu x b_1)\,([\![xx]\!]b_1 \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \stackrel{\Delta}{=}, \equiv & \\
(\nu x b_1)\,([\![xx]\!]b_1 \mid (\nu y b)\,(y(\circ).\,\overline{b}\langle\circ\rangle \mid \overline{x}\langle\langle y, b\rangle\rangle) \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \stackrel{\Delta}{=}, \equiv & \\
(\nu x b_1 c_1 y b)\,(x(\circ).\,\overline{c_1}\langle\circ\rangle \mid c_1(\langle b_2, d\rangle).\,(!\,[\![x]\!]b_2 \mid d{=}b_1) & & \\
\qquad \mid [\![y]\!]b \mid \overline{x}\langle\langle y, b\rangle\rangle \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \to_\pi & (x) \\
(\nu x b_1 c_1 y b)\,(\overline{c_1}\langle\langle y, b\rangle\rangle \mid c_1(\langle b_2, d\rangle).\,(!\,[\![x]\!]b_2 \mid d{=}b_1) & & \\
\qquad \mid [\![y]\!]b \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \to_\pi & (c_1) \\
(\nu x b_1 y b)\,(!\,[\![x]\!]y \mid b{=}b_1 \mid [\![y]\!]b \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \equiv & (\alpha) \\
(\nu x b_1 y b)\,(!\,[\![x]\!]y \mid b{=}b_1 \mid [\![y]\!]b & & \\
\qquad \mid (\nu z b_2)\,([\![z]\!]b_2 \mid \overline{x}\langle\langle z, b_2\rangle\rangle) \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \equiv &
\end{array}
$$

$$
\begin{array}{ll}
(\nu x b_1 y b z b_2)\,(x(\circ).\overline{y}\langle\circ\rangle \mid\, !\,[\![x]\!]y \mid b{=}b_1 \mid [\![y]\!]b & \\
\qquad\qquad \mid [\![z]\!]b_2 \mid \overline{x}\langle\langle z, b_2\rangle\rangle \mid\, !\,[\![\lambda y.y]\!]x \mid b_1{=}a) & \rightarrow_\pi^* (x, y, b, b_1) \\
(\nu x y z b_2)\,(!\,[\![x]\!]y \mid \overline{a}\langle\langle z, b_2\rangle\rangle \mid [\![z]\!]b_2 \mid !\,[\![\lambda y.y]\!]x) & \equiv \\
(\nu z b)\,([\![z]\!]b \mid \overline{a}\langle\langle z, b\rangle\rangle) & \triangleq\ [\![\lambda z.z]\!]a
\end{array}
$$

Notice that we performed the two substitutions without resorting to the renaming of outputs of encoded λ-terms, and that all those renamings take place at the end.

Notice also that, because the encoding implements a limited notion of substitution, the reduction does *not* run past

$$(\nu c)\,([\![\lambda y.y]\!]c \mid c(\langle b, d\rangle).\,(!\,[\![\lambda y.y]\!]b \mid d{=}a)) \triangleq [\![(\lambda y.y)(\lambda y.y)]\!]a.$$

The only expression that gets close is that in the sixth line, which corresponds to

$$(\nu x b_1 c_1 y b)\,([\![\lambda y.y]\!]c_1 \mid c_1(\langle b_2, d\rangle).\,(!\,[\![x]\!]b_2 \mid d{=}b_1) \mid !\,[\![\lambda y.y]\!]x \mid b_1{=}a)$$

which is (up to renaming) $[\![((\lambda y.y)x)\langle x = (\lambda y.y)\rangle]\!]a$. In fact, this is as expected, since:

$$(\lambda x.xx)(\lambda y.y) \rightarrow_{\mathbf{xs}} xx\langle x = (\lambda y.y)\rangle \rightarrow_{\mathbf{xs}} ((\lambda y.y)x)\langle x = (\lambda y.y)\rangle$$

We can also show that $(\nu x)\,([\![M]\!]a \mid !\,[\![N]\!]x)$ represents (implicit) term substitution successfully, at least for closed N.

Theorem 24. *For N a closed λ-term, and M any λ-term:*

$$(\nu x)\,([\![M]\!]a \mid !\,[\![N]\!]x) \succeq^{\pi} [\![M[N/x]]\!]a.$$

In [21], a similar result is shown, but for the *higher-order* π-calculus, where substitution is a primitive operation, so the encoding is easier.

That we had to limit the contraction of redexes $(\lambda x.M)N$ to those where N is closed, might seem a strong restriction. However, notice that Milner's simulation result is stated, not just for lazy reduction - which is a weaker notion than spine reduction - but also only for closed λ-terms. Consider a reduction $M \rightarrow_{\mathrm{L}} M'$, where M is closed, then $M = (\lambda x.M_1)NM_2\cdots M_n$; since M is closed, so is N, so our result is as strong as need be in the context of Milner's result.

Corollary 25. If $M \rightarrow_{\mathrm{L}} N$, and M is a closed λ-term, then $[\![M]\!]a \succeq^{\pi} [\![N]\!]a$.

5 Context assignment

The π-calculus is equipped with a rich type theory [23]: from the basic type system for counting the arity of channels to sophisticated linear types in [17], which studies a relation between Call-by-Value $\lambda\mu$ and a linear π-calculus. Linearisation is used to be able to achieve processes that are functions, by allowing output over one channel name only, in a λ-calculus, natural deduction style. Moreover, the encoding presented in [17] is type dependent, in that, for each term, different π-processes are assigned, depending on the original type; this makes the encoding quite cumbersome.

The type system presented in this section differs quite drastically from the standard type system presented in [23]: here input and output channels essentially have the type of the data they are sending or receiving, and are separated by the type system by putting

all inputs with their types on the left of the sequent, and the outputs on the right. In our system, types give a logical view to the π-calculus rather than an abstract specification on how channels should behave. Our encoding is very simple and intuitive by interpreting the cut operationally as a communication. The idea of giving a computational interpretation of the cut as a communication primitive is also used by [3] and [9]. In both papers, only a small fragment of Linear Logic was considered, and the encoding between proofs and π-calculus was left rather implicit.

In this section, we define a notion of context assignment for processes in π that describes the '*input-output interface*' of a process, by assigning a left-context, containing the types for the input channels, and a right-context, containing the types for the output channels; it was first presented in [4]. This notion is different from others in that it assigns to channels the type of the input or output that is sent over the channel.

Context assignment was defined in [4] to establish preservation of assignable types under the interpretation of the sequent calculus $\mathcal{X}$, as presented in [6], into the π-calculus. As for the notion of type assignment on $\mathcal{X}$ terms, in the typing judgements we write names used for input on the left and names used for output on the right; this implies that, if a name is both used to send and to receive, it will appear on both sides, with the same type. Since $\mathcal{X}$ offers a natural presentation of the classical propositional calculus with implication, and enjoys the Curry-Howard isomorphism for the implicative fragment of Gentzen's system LK [12], this implies that the notion of context assignment as defined below is *classical* (i.e. not intuitionistic) in nature.

We now repeat the definition of (simple) type assignment; we first define types and contexts.

Definition 26 (Types and Contexts).

1. The set of types is defined by the grammar: $A, B ::= \varphi \mid A \to B$, where φ is a basic type of which there are infinitely many.
2. An *input context* Γ is a mapping from names to types, denoted as a finite set of *statements* $a{:}A$, such that the *subject* of the statements (a) are distinct. We write Γ_1, Γ_2 to mean the *compatible union* of Γ_1 and Γ_2 (if Γ_1 contains $a{:}A_1$ and Γ_2 contains $a{:}A_2$, then $A_1 = A_2$), and write $\Gamma, a{:}A$ for $\Gamma, \{a{:}A\}$.
3. *Output* contexts Δ, and the notions Δ_1, Δ_2, and $n{:}A, \Delta$ are defined in a similar way.
4. If $n{:}A \in \Gamma$ and $n{:}B \in \Delta$, then $A = B$.

Definition 27 ((Classical) Context Assignment). Context assignment for the π-calculus with pairing is defined by the following sequent system:

$$(0): \frac{}{0 : \Gamma \vdash_\pi \Delta} \qquad (\textit{pair-out}): \frac{}{\overline{a}\langle\langle b, c\rangle\rangle : \Gamma, b{:}A \vdash_\pi a{:}A \to B, c{:}B, \Delta}$$

$$(!): \frac{P : \Gamma \vdash_\pi \Delta}{!\,P : \Gamma \vdash_\pi \Delta} \qquad (\textit{out}): \frac{}{\overline{a}\langle b\rangle : \Gamma, b{:}A \vdash_\pi a{:}A, b{:}A, \Delta}\ (a \neq b)$$

$$(\nu): \frac{P : \Gamma, a{:}A \vdash_\pi a{:}A, \Delta}{(\nu a)\,P : \Gamma \vdash_\pi \Delta} \qquad (\textit{let}): \frac{P : \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{\textit{let}\,\langle x, y\rangle = z\ \textit{in}\ P : \Gamma, z{:}A \to B \vdash_\pi \Delta}$$

$$(|): \frac{P : \Gamma \vdash_\pi \Delta \quad Q : \Gamma \vdash_\pi \Delta}{P \mid Q : \Gamma \vdash_\pi \Delta} \qquad (\textit{in}): \frac{P : \Gamma, x{:}A \vdash_\pi x{:}A.\Delta}{a(x).\,P : \Gamma, a{:}A \vdash_\pi \Delta}$$

The side-condition on rule (out) is there to block the derivation of $\overline{a}\langle a\rangle : \vdash_\pi a{:}A$.

Example 28. Although we have no rule $(pair\text{-}in)$, it is admissible, since we can derive

$$\dfrac{\dfrac{P : \Gamma, y{:}B \vdash_\pi x{:}A, \Delta}{let\,\langle x,y\rangle = z\ in\ P : \Gamma, z{:}A\to B \vdash_\pi \Delta}}{a(z).\,let\,\langle x,y\rangle = z\ in\ P : \Gamma, a{:}A\to B \vdash_\pi \Delta}$$

This notion of type assignment does not (directly) relate back to the logical calculus LK. For example, rules $(|)$ and $(!)$ do not change the contexts, so do not correspond to any rule in LK, not even to a $\lambda\mu$-style [19] activation step; moreover, rule (ν) just removes a formula.

The *weakening* rule is admissible:

$$(W) : \frac{P : \Gamma \vdash_\pi \Delta}{P : \Gamma' \vdash_\pi \Delta'}\ (\Gamma' \supseteq \Gamma, \Delta' \supseteq \Delta)$$

This result allows us to be a little less precise when we construct derivations, and allow for rules to join contexts, by using, for example, the rule

$$(|) : \frac{P : \Gamma_1 \vdash_\pi \Delta_1 \quad Q : \Gamma_2 \vdash_\pi \Delta_2}{P \mid Q : \Gamma_1, \Gamma_2 \vdash_\pi \Delta_1, \Delta_2}$$

so switching, without any scruples, to multiplicative style, whenever convenient.

We have a soundness (witness reduction) result for our notion of type assignment for π as shown in [4].

Theorem 29 (Witness reduction [4]). *If $P : \Gamma \vdash_\pi \Delta$ and $P \to_\pi Q$, then $Q : \Gamma \vdash_\pi \Delta$.*

We will now show that our interpretation preserves types assignable to lambda terms using Curry's system, which is defined as follows:

Definition 30 (Curry type assignment for the λ-calculus).

$$(Ax) : \frac{}{\Gamma, x{:}A \vdash_\lambda x : A} \qquad (\to I) : \frac{\Gamma, x{:}A \vdash_\lambda M : B}{\Gamma \vdash_\lambda \lambda x.M : A\to B}$$

$$(\to E) : \frac{\Gamma \vdash_\lambda M : A\to B \quad \Gamma \vdash_\lambda N : A}{\Gamma \vdash_\lambda MN : B}$$

We can now show that typeability is preserved by $[\![\cdot]\!]\cdot$:

Theorem 31. *If $\Gamma \vdash_\lambda M : A$, then $[\![M]\!]a : \Gamma \vdash_\pi a{:}A$.*

PROOF. By induction on the structure of derivations in $\vdash_\lambda$; notice that we use implicit weakening.

(Ax) : Then $M = x$, and $\Gamma = \Gamma', x{:}A$. Notice that $x(\circ).\overline{a}\langle\circ\rangle = [\![x]\!]a$, and that

$$\dfrac{\overline{\overline{a}\langle\circ\rangle : \Gamma, \circ{:}A \vdash_\pi a{:}A, \circ{:}A}}{x(\circ).\overline{a}\langle\circ\rangle : \Gamma', x{:}A \vdash_\pi a{:}A}$$

$(\rightarrow I)$: Then $M = \lambda x.N$, $A = C \rightarrow D$, and $\Gamma, x{:}C \vdash_\lambda N : D$. Then, by induction, a derivation $\mathcal{D} :: [\![N]\!] b : \Gamma, x{:}C \vdash_\pi b{:}D$ exists, and we can construct:

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\mathcal{D}}}{[\![N]\!] b : \Gamma, x{:}C \vdash_\pi b{:}D} \quad \dfrac{}{\overline{a}\langle\langle x, b\rangle\rangle : x{:}C \vdash_\pi a{:}C{\rightarrow}D, b{:}D}}{[\![N]\!] b \mid \overline{a}\langle\langle x, b\rangle\rangle : \Gamma, x{:}C \vdash_\pi a{:}C{\rightarrow}D, b{:}D}}{(\nu b)([\![N]\!] b \mid \overline{a}\langle\langle x, b\rangle\rangle) : \Gamma, x{:}C \vdash_\pi a{:}C{\rightarrow}D}}{(\nu x b)\,([\![N]\!] b \mid \overline{a}\langle\langle x, b\rangle\rangle) : \Gamma \vdash_\pi a{:}C{\rightarrow}D}$$

Notice that $(\nu x b)\,([\![N]\!] b \mid \overline{a}\langle\langle x, b\rangle\rangle) = [\![\lambda x.N]\!] a$.

$(\rightarrow E)$: Then $M = PQ$, and there exists B such that $\Gamma \vdash_\lambda P : B{\rightarrow}A$ and $\Gamma \vdash_\lambda Q : B$. By induction, there exist $\mathcal{D}_1 :: [\![P]\!] c : \Gamma \vdash_\pi c{:}B{\rightarrow}A$ and $\mathcal{D}_2 :: [\![Q]\!] b : \Gamma \vdash_\pi b{:}B$, and we can construct:

$$\dfrac{\dfrac{\boxed{\mathcal{D}_1}}{[\![P]\!] c : \Gamma \vdash_\pi c{:}B{\rightarrow}A} \quad \dfrac{\dfrac{\dfrac{\boxed{\mathcal{D}_2}}{[\![Q]\!] b : \Gamma \vdash_\pi b{:}B}}{!\,[\![Q]\!] b : \Gamma \vdash_\pi b{:}B} \quad \dfrac{}{\overline{a}\langle\circ\rangle : \Gamma, \circ{:}A \vdash_\pi a{:}A, \circ{:}A, \Delta}}{d{=}a : d{:}A \vdash_\pi a{:}A}}{!\,[\![Q]\!] b \mid d{=}a : \Gamma, d{:}A \vdash_\pi b{:}B, a{:}A}}{c(\langle b, d\rangle).\,(!\,[\![Q]\!] b \mid d{=}a) : \Gamma, c{:}B{\rightarrow}A \vdash_\pi a{:}A}}{[\![P]\!] c \mid c(\langle b, d\rangle).\,(!\,[\![Q]\!] b) \mid d{=}a) : \Gamma, c{:}B{\rightarrow}A \vdash_\pi c{:}B{\rightarrow}A, a{:}A}}{(\nu c)\,([\![P]\!] c \mid c(\langle b, d\rangle).\,(!\,[\![Q]\!] b \mid d{=}a)) : \Gamma \vdash_\pi a{:}A}$$

and $[\![PQ]\!] a = (\nu c)\,([\![P]\!] c \mid c(\langle b, d\rangle).\,(!\,[\![Q]\!] b \mid d{=}a))$. □

Notice that although, in the above proof, we are only interested in showing results with *one* typed output (conclusion) – after all, we are interpreting the typed λ-calculus, an intuitionistic system – we need the classical, multi-conclusion character of our type assignment system for π to achieve this result.

Conclusions and Future Work

We have found a new, simple and intuitive encoding of λ-terms in π that respects our definition of explicit spine reduction, is similar with normal reduction, and encompasses Milner's lazy reduction on closed terms. We have shown that, for our context assignment system that uses the type constructor $\rightarrow$ for π and is based on classical logic, assignable types for λ-terms are preserved by our interpretation as typeable π-processes. We managed this without having to linearise the calculus as done in [17].

The classical sequent calculus $\mathcal{X}$ has two natural, dual notions of sub-reduction, called Call-by-Name and Call-by-Value; we will investigate if the interpretation of these systems in to the π-calculus gives natural notions of CBN of CBV reduction on π-processes, and if this enables CBN or CBV logical encodings of the λ-calculus.

Acknowledgements

We would like to thank Jan Willem Klop, Fer-Jan de Vries, and Vincent van Oostrom for their willingness to instruct, and Martin Berger for useful comments.

References

1. M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *CC&CS'97*, 36–47, 1997.
2. S. Abramsky. The lazy lambda calculus. In *Research topics in functional programming*, 65–116, 1990.
3. Samson Abramsky. Proofs as Processes. *TCS*, 135(1):5–9, 1994.
4. S. van Bakel, L. Cardelli, and M.G. Vigliotti. From $\mathcal{X}$ to π; Representing the Classical Sequent Calculus in π-calculus. In *CL&C'08*, 2008.
5. S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and Classical Logic. In *ICTCS'05*, *LNCS* 3701, 81–96, 2005.
6. S. van Bakel and P. Lescanne. Computation with Classical Sequents. *MSCS*, 18:555–609, 2008.
7. H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.
8. H.P. Barendregt, R. Kennaway, J.W. Klop, and M.R. Sleep. Needed Reduction and Spine Strategies for the Lambda Calculus. *I&C*, 75(3):191–231, 1987.
9. G. Bellin and P.J. Scott. On the pi-Calculus and Linear Logic. *TCS*, 135(1):11–65, 1994.
10. R. Bloo and K.H. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *CSN'95 – Computer Science in the Netherlands*, 62–72, 1995.
11. A. Church. A note on the entscheidungsproblem. *JSL*, 1(1):40–41, 1936.
12. G. Gentzen. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*. Ed M. E. Szabo, North Holland, 68ff (1969), 1935.
13. G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935.
14. J. Goubault-Larrecq. A Few Remarks on SKInT. RR-3475, INRIA Rocquencourt, 1998.
15. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, *LNCS* 512, 133–147, 1991.
16. K. Honda and N. Yoshida. On the Reduction-based Process Semantics. *TCS*, 151:437–486, 1995.
17. K. Honda, N. Yoshida, and M. Berger. Control in the π-Calculus. In *CW'04*, 2004.
18. R. Milner. Function as processes. *MSCS*, 2(2):269–310, 1992.
19. M. Parigot. An algorithmic interpretation of classical natural deduction. In *LPAR'92*, *LNCS* 624, 190–201, 1992.
20. D. Sangiorgi. *Expressing Mobility in Process Algebra: First Order and Higher order Paradigms*. PhD thesis, Edinburgh University, 1992.
21. D. Sangiorgi. An Investigation into Functions as Processes. In *MFPS'93*, 143–159, 1993.
22. D. Sangiorgi. Lazy functions and processes. RR2515, INRIA, Sophia-Antipolis, 1995.
23. D. Sangiorgi and D. Walker. *The Pi-Calculus*. Cambridge University Press, 2003.
24. P. Sestoft. Standard ML on the Web server. Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark, 1996.
25. H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.
26. C. Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, 2000.
27. C. Urban and G. M. Bierman. Strong normalisation of cut-elimination in classical logic. *FI*, 45(1,2):123–155, 2001.
28. F.-J. de Vries. Böhm trees, bisimulations and observations in lambda calculus. In *FLP'97*, 230–245, 1997.