Asynchronous π-calculus at Work: The Call-by-Need Strategy

Davide Sangiorgi(✉)

Focus Team, University of Bologna and Inria, Bologna, Italy
davide.sangiorgi@gmail.com

Abstract. In a well-known and influential paper [17] Palamidessi has shown that the expressive power of the Asynchronous π-calculus is strictly less than that of the full (synchronous) π-calculus. This gap in expressiveness has a correspondence, however, in sharper semantic properties for the former calculus, notably concerning algebraic laws. This paper substantiates this, taking, as a case study, the encoding of call-by-need λ-calculus into the π-calculus. We actually adopt the *Local* Asynchronous π-calculus, that has even sharper semantic properties. We exploit such properties to prove some instances of validity of β-reduction (meaning that the source and target terms of a β-reduction are mapped onto behaviourally equivalent processes). Nearly all results would fail in the ordinary synchronous π-calculus. We show that however the full β-reduction is not valid. We also consider a refined encoding in which some further instances of β-validity hold. We conclude with a few questions for future work.

1 Introduction

Since the introduction of the π-calculus, a lot of effort has been devoted to its comparison with the λ-calculus, beginning with Milner's seminar work on functions as processes [14]. The attention has gone mostly to *call-by-name* and *call-by-value* λ-calculi [19], and the main results concern operational correspondence, validity of β-reduction, characterisation of the equivalence induced on λ-terms by the π-calculus encoding [6,14,21,22,27]. In particular, the call-by-name encoding, for its simplicity, is often presented as *the* π-calculus representation of functions.

In a call-by-name reduction, the redex contracted is the leftmost one; the reduction occurs regardless of whether the argument of the function is a value (as in call-by-value). As a consequence, if the argument is not a value and will be used several times, its evaluation will be repeated the same number of times. In implementation of programming languages following call-by-name, this repetition of evaluation is avoided: evaluation occurs only once, the first time the term is used, and the value so obtained is recorded for future uses. This implementation technique is referred to as *call-by-need* evaluation (or strategy) [28]. Thus call-by-need uses explicit environments and β-reduction does not require

M. S. Alvim et al. (Eds.): Palamidessi Festschrift, LNCS 11760, pp. 33–49, 2019.
https://doi.org/10.1007/978-3-030-31175-9_3

substituting a term for a variable, as in call-by-name (or call-by-value)—just substituting a reference to a term for a variable. In this sense call-by-need is closer to the π-calculus than call-by-name, as substitutions in the π-calculus only involve names. Again, the modifications that take us from call-by-name to call-by-need can be easily represented in a π-calculus encoding [24].

The π-calculus, having a rich and well-developed theory, as well as a remarkable expressiveness, has been advocated as a foundational model for reasoning about higher-order languages, including equivalence between programs and correctness of compilers and compiler optimisations [25,26]. Indeed, the π-calculus and related languages have been used, via appropriate encodings, as a target language of compilers, for a number of experimental programming languages, beginning with Pict [18] and Join [7].

The above raises the question about how, and at which extent, the π-calculus and its current theory can be used to prove the correctness of call-by-need as an optimised implementation strategy for call-by-name. The only work on the correctness of the π-calculus representation of call-by-need is by Brock and Ostheimer [5]. The paper considers operational correspondence, between reduction in a call-by-need system and in the encoding π-calculus terms. However there are foundametal semantic issues that remain unexplored. A major one is the validity of β-reduction, namely the property that the processes encoding β-convertible λ-terms are behaviourally iundistinguishable. The property holds in call-by-name (and it is at the heart of its theory), as well as in the π-calculus encoding of call-by-name. One would therefore hope to find analoguos results for call-by-need. The correctness of the process representation of call-by-need is the topic of the present paper, focusing on the validity of β-reduction.

In a well-known and influential paper [17] Palamidessi has shown that the expressive power of the asynchronous π-calculus is strictly less than that of the full (synchronous) π-calculus. This gap in expressiveness has a correspondence, however, in sharper semantic properties for the former calculus, notably concerning algebraic laws. This paper may be seen as a demonstration of this, since most the proofs are carried out using algebraic laws that are only valid in the asynchronous π-calculus—precisely in the Asynchronous Local π-calculus, ALπ, [12], where only the output capability of names may be exported.

In Sect. 2 we present ALπ and some of its laws. In Sect. 3 we briefly recall the call-by-name and call-by-need λ-calculus. In Sect. 4 we consider two encodings of call-by-need. We show that limited forms of validity β-reduction hold, and that the general property fails. The questions that follow from this, discussed in Sect. 5, may contribute to open some interesting directions for future work, which may also shed further light on the theory of the π-calculus and similar name-passing calculi.

2 The Asynchronous Local π-calculus

2.1 Syntax

Small letters $a, b, \ldots, x, y, \ldots$ range over the infinite set of names, and $P, Q, R, \ldots$ over the set of all processes. A tilde represents a tuple. The i-th elements of a tuple $\widetilde{E}$ is referred to as $\widetilde{E}_i$. Our notations are extended to tuples componentwise.

The Asynchronous Local π-calculus (ALπ) [12] is built from the operators of inaction, input prefix, output, parallel composition, restriction, and replication:

$$P := \mathbf{0} \;\Big|\; a(\widetilde{b}).P \;\Big|\; \overline{a}\langle\widetilde{b}\rangle \;\Big|\; P_1 \mid P_2 \;\Big|\; \boldsymbol{\nu} a\, P \;\Big|\; !a(\widetilde{b}).P\,.$$

with the *syntactic constraint* that in processes $a(\widetilde{b}).P$ and $!a(\widetilde{b}).P$ names $\widetilde{b}$ may not occur free in P in input position.

When the tilde is empty, the surrounding brackets $()$ and $\langle\rangle$ will be omitted. $\mathbf{0}$ is the inactive process. An input-prefixed process $a(\widetilde{b}).P$, where $\widetilde{b}$ has pairwise distinct components, waits for a tuple of names $\widetilde{c}$ to be sent along a and then behaves like $P\{\widetilde{c}/\widetilde{b}\}$, where $\{\widetilde{c}/\widetilde{b}\}$ is the simultaneous substitution of names $\widetilde{b}$ with names $\widetilde{c}$. An output particle $\overline{a}\langle\widetilde{b}\rangle$ emits names $\widetilde{b}$ at a. Parallel composition is to run two processes in parallel. The restriction $\boldsymbol{\nu} a\, P$ makes name a local, or private, to P. A replication $!P$ stands for a countable infinite number of copies of P in parallel. We assign parallel composition the lowest precedence among the operators.

2.2 Terminologies and Notations

We write $\overline{a}(b).b(\widetilde{c}).Q$ as an abbreviation for $\boldsymbol{\nu} b\, (\overline{a}b \mid b(\widetilde{c}).Q)$, and similarly for $\overline{a}(b).!b(\widetilde{c}).Q$. The prefix '$\overline{a}(b)$' is called a *bound output*. In prefixes $a(\widetilde{b})$ and $\overline{a}\langle\widetilde{b}\rangle$, we call a the *subject* and $\widetilde{b}$ the *object*. We use α to range over prefixes. We often abbreviate $\alpha.\mathbf{0}$ as α, and $\boldsymbol{\nu} a\, \boldsymbol{\nu} b\, P$ as $(\boldsymbol{\nu} a, b)\, P$. An input prefix $a(\widetilde{b}).P$ and a restriction $\boldsymbol{\nu} b\, P$ are binders for names $\widetilde{b}$ and b, respectively, and give rise in the expected way to the definition of *free names* (fn), *bound names* (bn) and *names* (n) of a term or a prefix, and *alpha conversion.* We identify processes or actions that only differ on the choice of the bound names. The symbol $=$ will mean "syntactic identity modulo alpha conversion". Sometimes, we use $\stackrel{\text{def}}{=}$ as abbreviation mechanism, to assign a name to an expression to which we want to refer later. In a statement, a name declared *fresh* is supposed to be different from any other name appearing in the objects of the statement, like processes or substitutions. Substitutions are of the form $\{\widetilde{b}/\widetilde{c}\}$, and are finite assignments of names to names. A context is a process expression with a hole $[\cdot]$ in it. We use C to range over contexts; then $C[P]$ is the process obtained from C by filling its hole with P.

2.3 Sorting

Following Milner [13], we only admit *well-sorted agents*, that is agents obeying a predefined *sorting* discipline in their manipulation of names. The sorting prevents arity mismatching in communications, like in $\overline{a}\langle b, c\rangle \mid a(x).Q$. A sorting is an assignment of *sorts* to names, which specifies the arity of each name and, recursively, of the names carried by that name. We do not present the formal system of sorting because it is not essential in the exposition of the topics in the present paper.

We will however allow sorting to identify *linear* names, that is, names that are supposed to be used only once. Linearity will be used a few times to replace input-replicated prefixes with ordinary input prefixes. Again, we omit the details of linearity in type systems (sorting is a form of type system), as they are by now standard [9,24].

$$\textsc{inp}:\ a(\widetilde{b}).\,P \xrightarrow{a(\widetilde{b})} P \qquad\qquad \textsc{out}:\ \overline{a}\langle\widetilde{b}\rangle \xrightarrow{\overline{a}\langle\widetilde{b}\rangle} \mathbf{0}$$

$$\textsc{rep}:\ \frac{P \mid !P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \qquad\qquad \textsc{par}:\ \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ \text{if } \mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset$$

$$\textsc{com}:\ \frac{P \xrightarrow{a(\widetilde{c})} P' \qquad Q \xrightarrow{(\boldsymbol{\nu}\,\widetilde{d})\,\overline{a}\langle\widetilde{b}\rangle} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\widetilde{d}\,(P'\{\widetilde{b}/\widetilde{c}\} \mid Q')}\ \text{if } \widetilde{d} \cap \mathsf{fn}(P) = \emptyset$$

$$\textsc{res}:\ \frac{P \xrightarrow{\mu} P'}{\boldsymbol{\nu}a\,P \xrightarrow{\mu} \boldsymbol{\nu}a\,P'}\ a \notin \mathsf{n}(\mu) \qquad \textsc{open}:\ \frac{P \xrightarrow{(\boldsymbol{\nu}\,\widetilde{d})\,\overline{a}\langle\widetilde{b}\rangle} P'}{\boldsymbol{\nu}c\,P \xrightarrow{(\boldsymbol{\nu}\,c,\widetilde{d})\,\overline{a}\langle\widetilde{b}\rangle} P'}\ c \in \widetilde{b} - \widetilde{d},\ a \neq c.$$

Fig. 1. The transition system for ALπ

2.4 Relations

A process has three possible forms of action. A *silent action* $P \xrightarrow{\tau} P'$ represents an interaction, i.e. an internal activity in P. *Input* and *output actions* are, respectively, of the form $P \xrightarrow{a(\widetilde{d})} P'$ and $P \xrightarrow{(\boldsymbol{\nu}\,\widetilde{d})\,\overline{a}\langle\widetilde{b}\rangle} P'$. In both cases, the action occurs at a—the *subject* of the action. In the output action, $\widetilde{b}$ is the tuple of names which are emitted, and $\widetilde{d} \subseteq \widetilde{b}$ are private names which are carried out from their current scope. We use μ to represent the label of a generic action (not to be confused with α, which represents prefixes). In an input action $a(\widetilde{d})$ and in an output action $(\boldsymbol{\nu}\,\widetilde{d})\,\overline{a}\langle\widetilde{b}\rangle$, names $\widetilde{d}$ are *bound*, the remaining ones free. Bound and free names of an action μ, respectively written $\mathsf{bn}(\mu)$ and $\mathsf{fn}(\mu)$, are defined accordingly. The *names* of μ, briefly $\mathsf{n}(\mu)$, are $\mathsf{bn}(\mu) \cup \mathsf{fn}(\mu)$. The transition system of the calculus is presented in Fig. 1. We have omitted the symmetric versions of rules PAR and COM. Alpha convertible processes have deemed to have the same transitions. We often abbreviate $P \xrightarrow{\tau} Q$ with $P \longrightarrow Q$. The 'weak' arrow $\Longrightarrow$ is the reflexive and transitive closure of $\longrightarrow$.

We use the symbol $\equiv$ to denote *structural congruence*, a relation used to rearrange the structure of processes [13]. We shall also use it to represent garbage-collection of restrictions and of inert terms.

Definition 1 (Structural congruence). Structural congruence*, $\equiv$, is the smallest congruence relation satisfying the axioms below:*

- $P \mid \mathbf{0} \equiv P,\ \ P \mid Q \equiv Q \mid P,\ \ P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
- $!a(x).P \equiv a(x).P \mid !a(x).P;$
- $\boldsymbol{\nu} a\, \mathbf{0} \equiv \mathbf{0},\ \ \boldsymbol{\nu} a\, \boldsymbol{\nu} b\, P \equiv \boldsymbol{\nu} b\, \boldsymbol{\nu} a\, P,\ \ \boldsymbol{\nu} a\, (P \mid Q) \equiv P \mid \boldsymbol{\nu} a\, Q$ *if* $a \notin \mathsf{fn}(P);$
- $\boldsymbol{\nu} a\, (P \mid !a(\widetilde{b}).Q) \equiv P$ *and* $\boldsymbol{\nu} a\, (P \mid a(\widetilde{b}).Q) \equiv P$*, if* $a \notin \mathsf{fn}(P)$.

(A derivable law is $\boldsymbol{\nu} a\, P \equiv P$, for a not free in P.) A standard behavioural equivalence for the π-calculus is *barbed congruence*. Barbed congruence can be defined in any calculus possessing: (i) an *interaction relation* (the τ-steps in the π-calculus), modelling the evolution of the system; and (ii) an *observability predicate* $\downarrow_a$ for each name a, which detects the possibility of a process of accepting a communication with the environment at a. More precisely, we write $P \downarrow_a$ if P can make an output action whose subject is a, that is, if there are P' and an output action μ with subject a such that $P \xrightarrow{\mu} P'$. We write $P \Downarrow_a$ if $P \Longrightarrow P'$ and $P' \downarrow_a$. Unlike synchronous π-calculus, in asynchronous calculi it is natural to restrict the observation to output actions [1]. The reason is that in asynchronous calculi the observer has no direct way of knowing when a message emitted is received.

Definition 2 (Barbed congruence). *A symmetric relation $\mathcal{S}$ on π-calculus processes is a* barbed bisimulation *if $P\ \mathcal{S}\ Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $P'\ \mathcal{S}\ Q'$.*
2. *If $P \downarrow_a$ then $Q \Downarrow_a$.*

Two π-calculus processes P and Q are barbed bisimilar *if $P\ \mathcal{S}\ Q$ for some barbed bisimulation $\mathcal{S}$. We say that P and Q are* barbed congruent*, written $P \approx Q$, if for each π-calculus context C, processes $C[P]$ and $C[Q]$ are barbed bisimilar.*

Strong barbed congruence, written $\sim$, is defined analogously but replacing the weak arrows $\Longrightarrow$ and $\Downarrow_a$ with the strong arrows $\longrightarrow$ and $\downarrow_a$. As expected, we have $\equiv\ \subseteq\ \sim\ \subseteq\ \approx$; each containment is strict.

2.5 Further Algebraic Laws

Most of the proofs in the paper are carried out using algebraic reasoning. We report here some important laws. First some simple laws that are valid in the full (synchronous) π-calculus (Lemma 1). Then laws that are specific to ALπ.

Lemma 1. *1.* $\boldsymbol{\nu} a\ (\overline{a}(b).P \mid a(y).Q) \ \approx\ (\boldsymbol{\nu} a, b)\ (P \mid Q\{b/y\})$ *and* $\boldsymbol{\nu} a\ (\overline{a}(b).P \mid !a(y).Q) \approx (\boldsymbol{\nu} a, b)\ (P \mid Q\{b/y\} \mid !a(y).Q);$

2. $\boldsymbol{\nu}a\,(\alpha.Q \mid !a(\widetilde{x}).P) \sim \alpha.\boldsymbol{\nu}a\,(Q \mid !a(\widetilde{x}).P)$, *if* $\mathsf{bn}(\alpha) \cap \mathsf{fn}(a(\widetilde{x}).P) = \emptyset$ *and* $a \notin \mathsf{n}(\alpha)$ *(a similar law holds without the replication);*
3. $a(\widetilde{x}).(P \mid !a(\widetilde{x}).P) \sim !a(\widetilde{x}).P$.

Important laws of ALπ are the following ones. Their validity hinges on the asynchronous and output-capability properties of ALπ. For simplicity we present them on monadic prefixes.

Lemma 2. *We have* $\overline{a}b \approx \boldsymbol{\nu}c\,(\overline{a}c \mid !c(x).\overline{b}x)$. *Moreover, if* b *is linear, then the replication can be removed thus:* $\overline{a}b \approx \boldsymbol{\nu}c\,(\overline{a}c \mid c(x).\overline{b}x)$.

Next, we report some distributivity laws for private replications, i.e., for systems of the form

$$\boldsymbol{\nu}y\,(P \mid !y(\widetilde{q}).Q)$$

in which y may occur free in P and Q only in output position. One should think of Q as a private resource of P, for P is the only process who can access Q; indeed P can activate as many copies of Q as needed. One such law has already been given as Lemma 1(2). (The laws can be generalised to the full π-calculus, but need stronger assumptions.)

Lemma 3. *Suppose* a *occurs free in* P, R, Q *only in output position. Then:*

1. $\boldsymbol{\nu}a\,(P \mid R \mid !a(\widetilde{b}).Q) \sim \boldsymbol{\nu}a\,(P \mid !a(\widetilde{b}).Q) \mid \boldsymbol{\nu}a\,(R \mid !a(\widetilde{b}).Q)$;
2. $\boldsymbol{\nu}a\,((!P) \mid !a(\widetilde{b}).Q) \sim !\boldsymbol{\nu}a\,(P \mid !a(\widetilde{b}).Q)$;
3. $\boldsymbol{\nu}a\,(\alpha.P \mid !a(\widetilde{b}).Q) \sim \alpha.\boldsymbol{\nu}a\,(P \mid !a(\widetilde{b}).Q)$, *if* $\mathsf{bn}(\alpha) \cap \mathsf{fn}(a(\widetilde{b}).Q) = \emptyset$ *and* $a \notin \mathsf{n}(\alpha)$;
4. $\boldsymbol{\nu}a\,((\boldsymbol{\nu}c\,P) \mid !a(\widetilde{b}).Q) \sim \boldsymbol{\nu}c\,\boldsymbol{\nu}a\,(P \mid !a(\widetilde{b}).Q)$ *if* $c \notin \mathsf{fn}(a(\widetilde{b}).Q)$.

ALπ has also sharper properties concerning labelled characterisation of bisimilarity and associated congruence properties [3,12].

3 The $\boldsymbol{\lambda}$-calculus

We use M, N to range over the set Λ of λ-terms, and x, y, z to range over variables. The set Λ of λ-terms is given by the grammar:

$$M ::= x \;\Big|\; \lambda x.M \;\Big|\; MN$$

A *redex* is a term of the form $(\lambda x.M)N$, and then its *contractum* is $M\{^{N}\!/\!_{x}\}$. In *call-by-name evaluation* [19], the redex is always at the extreme left of a term. We omit the standard evaluation rules.

Call-by-need [2,28] optimises call-by-name as follows, so to guarantee that in the contractum $M\{^{N}\!/\!_{x}\}$ the evaluation of N is not performed more than once. Roughly, N is placed in an environment, and the evaluation continues on M. When x is needed (i.e., x reaches the leftmost position), then N is evaluated and, if a value (i.e., an abstraction) is obtained, say V, then V replaces x (value V can replace all occurrences of x or, more commonly, only the leftmost occurrence, and then other occurrences of x when they reach the outermost position). Call-by-need is best presented in a graph; or in a system with a `let` construct to represent sharing. We refer to Ariola et al. [2] for details, as they are not essential for understanding the remainder of the paper; see also the references in Sect. 5.

We sometimes omit λ in nested abstractions, thus for example, $\lambda x_1 x_2.M$ stands for $\lambda x_1.\lambda x_2.M$. We assume the standard concepts of free and bound variables and substitutions, and identify α-convertible terms. Thus, throughout the paper '$=$' is syntactic equality modulo α-conversion.

Following the call-by-value terminology, the set of abstractions and variables are the *values*. (Indeed, call-by-need may also be thought of as a modified form of call-by-value, in which the evaluation of the argument of a function $\lambda x.M$ is made only when x is used for the first time, rather than before performing the reduction.)

4 The Encoding and Its Properties

4.1 Background Material

Figure 2 presents the call-by-name and call-by-need encodings [16,24]. The call-by-name one is a variant of the original encoding by Milner [14], with the advantage that it can be written in ALπ and can be easily modified to follow call-by-need.

We explain the encodings. The important part is the treatment of application. Both in call-by-name and in call-by-need, a function located at q (its 'location') is a process that signals to be a function on q, and then receives a pointer x to the argument N together with the location p for the next interaction. Now the evaluation of M continues. The difference between call-by-name and call-by-need arises when the argument N is needed. This is signaled by an output at x that also provides the location for the evaluation of a copy of N. In call-by-name, every output at x triggers the evaluation of a new copy of N. In call-by-need, in contrast, the evaluation is made only the first time. Precisely, in call-by-need N is evaluated at the first request and, when it becomes a value, a pointer to this value is returned (instantiating w, in the table). This pointer is returned to the process that requested N. When further requests for N are made, the pointer is returned immediately. Thus, for instance, in the call-by-name encoding of $(\lambda.xx)(II)$ term II is evaluated twice, whereas in the call-by-need encoding only once. In all encodings, the location names (in the table, the names ranged over by p, q, r) are linear.

Correcteness of call-by-name has been studied in depth. In particular, it has been shown that β-reduction is validated by the encoding, that the encoding gives rise to a term model for the λ-calculus, and that the equivalence on λ-terms induced by the encoding corresponds to the best tree-structures of the λ-calculus—which are also at the heart of its denotational semantics—namely Böhm Trees and Lévy-Longo Trees [14,23] Correctness of the call-by-need encoding has been studied only by Brock and Ostheimer [5], and only for operational correspondence with respect to Ariola et al.'s system [2]. (The encoding in Fig. 2

call-by-name encoding

$$\mathcal{M}[\![\lambda x.\, M]\!]p \overset{\text{def}}{=} \overline{p}(v).\, !v(x,q).\, \mathcal{M}[\![M]\!]q$$

$$\mathcal{M}[\![x]\!]p \overset{\text{def}}{=} \overline{x}p$$

$$\mathcal{M}[\![MN]\!]p \overset{\text{def}}{=} (\boldsymbol{\nu} q\,)\Big(\mathcal{M}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.\, !x(r).\, \mathcal{M}[\![N]\!]r\Big)$$

call-by-need encoding

$$\mathcal{N}[\![\lambda x.\, M]\!]p \overset{\text{def}}{=} \overline{p}(v).\, !v(x,q).\, \mathcal{N}[\![M]\!]q$$

$$\mathcal{N}[\![x]\!]p \overset{\text{def}}{=} \overline{x}p$$

$$\mathcal{N}[\![MN]\!]p \overset{\text{def}}{=}$$

$$(\boldsymbol{\nu} q\,)\Big(\mathcal{N}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.\, x(r).\, \boldsymbol{\nu} q'\, (\mathcal{N}[\![N]\!]q' \mid q'(w).\, (\overline{r}w \mid !x(r').\, \overline{r'}w))\Big)$$

Fig. 2. The encoding of call-by-name and call-by-need

is actually a minor improvement over that in [5]—avoiding one reduction step during a β-reducition—and maintains the results of operational correspondence in [5] recalled below.) Following Ariola et al.'s system [2] we write $M \Downarrow$ if the call-by-need computation of M terminates, and $M \Uparrow$ it the computation does not terminate.

Theorem 1 (Brock and Ostheimer [5]). *We have, for M closed:*

1. $M \Downarrow$ *iff* $\mathcal{N}[\![M]\!]p \Downarrow_p$;
2. $M \Uparrow$ *iff* $\mathcal{N}[\![M]\!]p \Uparrow$.

The proof in [5] considers an extended version of the call-by-need system in [2], one that yields a closer (nearly one-to-one) correspondence between reductions in the call-by-need system and reductions on the encoding π-calculus processes.

Note that, since M is closed, the only free name of $\mathcal{N}[\![M]\!]p$ is p; and since p is used in $\mathcal{N}[\![M]\!]p$ in output, the first visible action of $\mathcal{N}[\![M]\!]p$ (if there is one) is an output at p.

However, operational correspondence alone is not fully satisfactory as a criterium for correctness. It does not ensure foundamental semantic properties of the source language terms. In the following sections we focus on the validity of β-reduction.

4.2 β-validity

We consider in this section a few cases of validity of β-reduction; that is, the property that a β-redex $(\lambda x.M)N$ and its contractum $M\{^N\!/\!x\}$ are barbed congruent when represented as ALπ processes.

A form of β-reduction that is straightforward to handle is one in which the argument is never used.

Theorem 2. *If $x \notin \mathsf{fv}(M)$ then $\mathcal{N}[\![(\lambda x.M)N]\!]p \approx \mathcal{N}[\![M\{^N\!/\!_x\}]\!]p$.*

A more interesting form deals with β-reduction between closed values.

Theorem 3. $\mathcal{N}[\![(\lambda x.M)(\lambda y.N)]\!]p \approx \mathcal{N}[\![M\{^{\lambda y.N}\!/\!_x\}]\!]p$.

Proof. Using algebraic reasoning, we first derive:

$$\begin{aligned}
&\mathcal{N}[\![(\lambda x.M)(\lambda y.N)]\!]p\\
&= (\boldsymbol{\nu} q\,)\Big(\mathcal{N}[\![\lambda x.M]\!]q \mid \\
&\qquad q(v).\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\boldsymbol{\nu} q'\,(\mathcal{N}[\![\lambda y.N]\!]q' \mid \\
&\qquad\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
&= (\boldsymbol{\nu} q\,)\Big(\overline{q}(v).!v(x,q').\mathcal{N}[\![M]\!]q' \mid \\
&\qquad q(v).\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\boldsymbol{\nu} q'\,(\mathcal{N}[\![\lambda y.N]\!]q' \mid \\
&\qquad\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
&\approx (\boldsymbol{\nu} x\,)\Big(\mathcal{N}[\![M]\!]p \mid \\
&\qquad x(r).\boldsymbol{\nu} q'\,(\mathcal{N}[\![\lambda y.N]\!]q' \mid \\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
&= (\boldsymbol{\nu} x\,)\Big(\mathcal{N}[\![M]\!]p \mid \\
&\qquad x(r).\boldsymbol{\nu} q'\,(\overline{q'}(v).!v(y,q'').\mathcal{N}[\![N]\!]q'' \mid \\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
&\approx (\boldsymbol{\nu} x\,)\Big(\mathcal{N}[\![M]\!]p \mid \\
&\qquad x(r).\boldsymbol{\nu} v\,(\,!v(y,q'').\mathcal{N}[\![N]\!]q'' \mid \\
&\qquad\qquad (\overline{r}v \mid !x(r').\overline{r'}v))\Big)\\
&\sim (\boldsymbol{\nu} x,v\,)\Big(\mathcal{N}[\![M]\!]p \mid \\
&\qquad !v(y,q'').\mathcal{N}[\![N]\!]q'' \mid \\
&\qquad x(r).(\overline{r}v \mid !x(r').\overline{r'}v)\Big)\\
&\sim (\boldsymbol{\nu} x,v\,)\Big(\mathcal{N}[\![M]\!]p \mid \\
&\qquad !v(y,q'').\mathcal{N}[\![N]\!]q'' \mid \\
&\qquad !x(r').\overline{r'}v\Big)
\end{aligned}$$

where the two occurrences of $\approx$ represent applications of law (1) of Lemma 1, and the two occurrences of $\sim$ are due to laws (2) and (3) of the same lemma, respectively.

Now we proceed by induction on the structure of M. If M is variable different from x then the two replications at v and x can be garbage-collected and we are done. If $M = x$, then

$$\begin{aligned}(\boldsymbol{\nu} x, v\,)(\mathcal{N}[\![M]\!]p | !v(y,q'').\mathcal{N}[\![N]\!]q'' | !x(r').\overline{r'}v) &=\\ (\boldsymbol{\nu} x, v\,)(\overline{x}p | !v(y,q'').\mathcal{N}[\![N]\!]q'' | !x(r').\overline{r'}v) &\approx\\ (\boldsymbol{\nu} x, v\,)(\overline{p}v | !v(y,q'').\mathcal{N}[\![N]\!]q'' | !x(r').\overline{r'}v) &\equiv\\ (\boldsymbol{\nu} v\,)(\overline{p}v | !v(y,q'').\mathcal{N}[\![N]\!]q'') &=\\ \mathcal{N}[\![\lambda y.N]\!]p\end{aligned}$$

where $\approx$ is obtained from law (1) of Lemma 1, and $\equiv$ from the garbage-collection laws of Definition 1.

When M is an abstraction or an application, we proceed by induction and exploit the distributivity properties of private replications in Lemma 3.

Finally we consider the case when the argument of the function is divergent—a form of β-reduction that is not valid in call-by-value.

Theorem 4. *Suppose* $(\lambda x.M)N$ *is closed. If* $N \Uparrow$ *then we have* $\mathcal{N}[\![(\lambda x.M)N]\!]p \approx \mathcal{N}[\![M\{^N\!/\!x\}]\!]p$.

Proof. Using Theorem 1 we have $\mathcal{N}[\![N]\!]q \Uparrow$, for any q, hence $\mathcal{N}[\![N]\!]q \approx \mathbf{0}$. As a consequence, using algebraic reasoning similar to that in the proof of Theorem 3, we obtain

$$\mathcal{N}[\![(\lambda x.M)N]\!]p \approx \boldsymbol{\nu} x\ (\mathcal{N}[\![M]\!]p \mid x(r).\mathbf{0})$$

Now, since x occurs in $\mathcal{N}[\![M]\!]p$ only in output subject position, each output at x, say $\overline{x}r$, can be removed, or replaced by $\mathcal{N}[\![N]\!]r$ (because in the relation $\approx$ with $\mathbf{0}$), up-to $\approx$. This yields $\mathcal{N}[\![M\{^N\!/\!x\}]\!]p$.

4.3 Failure of General β-validity

However, in call-by-name and call-by-need β-reduction is not confined to values. We show that, in the call-by-need encoding, the general β-reduction fails. Notably β-reduction fails when the argument of the function is a variable. For this we show that

$$\mathcal{N}[\![yy]\!]p \not\approx \mathcal{N}[\![(\lambda z.zz)y]\!]p \tag{1}$$

While for simplicity this counterexample is shown for open terms, a similar one can be given for closed terms, by closing the two terms with an abstraction, i.e.,

$$\mathcal{N}[\![\lambda y.(yy)]\!]p \not\approx \mathcal{N}[\![\lambda y.((\lambda z.zz)y)]\!]p$$

The remainder of the section is devoted to the proof of (1). We first unroll the initial traces of the two processes (the only traces that they can perform) We have:

$$\begin{aligned}&\mathcal{N}[\![yy]\!]p\\ &= (\boldsymbol{\nu} q'\,)\Big(\overline{y}q' \mid q'(w).\boldsymbol{\nu} x\,\overline{w}\langle x',p\rangle.x'(r).\boldsymbol{\nu} q''\ (\overline{y}q'' \mid\\ &\qquad\qquad q''(w').(\overline{r}w' \mid !x(r').\overline{r'}w'))\Big)\\ &\xrightarrow{\overline{y}(q')}\xrightarrow{q'(w)}\xrightarrow{\boldsymbol{\nu} x'\ \overline{w}\langle x',p\rangle}\xrightarrow{x'(r)} \boldsymbol{\nu} q''\ (\overline{y}q'' \mid\\ &\qquad\qquad q''(w').(\overline{r}w' \mid !x'(r').\overline{r'}w'))\\ &\xrightarrow{\overline{y}(q'')}\xrightarrow{q''(w')}\qquad \overline{r}w' \mid !x'(r').\overline{r'}w'\end{aligned}$$

Since the above is the only possible trace of the term, we have

$$\mathcal{N}[\![yy]\!]p \sim \overline{y}(q).q(v).\boldsymbol{\nu}x\, \overline{v}\langle x,p\rangle.x(r).\overline{y}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \tag{2}$$

We now consider the analogous trace for $\mathcal{N}[\![(\lambda z.zz)y]\!]p$. Below, the uses of $\equiv$ are due to some garbage-collection of restrictions and private inputs (possibly replicated), and some rearrangements of the scopes of some restrictions; the use of $\sim$ is due to (2).

$$\begin{aligned}
&\mathcal{N}[\![(\lambda z.zz)y]\!]p\\
=\ &(\boldsymbol{\nu}q\,)\Big(\mathcal{N}[\![\lambda z.zz]\!]q \mid\\
&\quad q(v).\boldsymbol{\nu}x\,\overline{v}\langle x,p\rangle.x(r).\boldsymbol{\nu}q'\,(\mathcal{N}[\![y]\!]q' \mid\\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
=\ &(\boldsymbol{\nu}q\,)\Big(\overline{q}(v).!v(z,q').\mathcal{N}[\![zz]\!]q' \mid\\
&\quad q(v).\boldsymbol{\nu}x\,\overline{v}\langle x,p\rangle.x(r).\boldsymbol{\nu}q'\,(\overline{y}q' \mid\\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
\xrightarrow{\tau}\xrightarrow{\tau}\equiv\ &\boldsymbol{\nu}x\,\Big(\mathcal{N}[\![xx]\!]p \mid\\
&\quad x(r).\boldsymbol{\nu}q'\,(\overline{y}q' \mid\\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
\sim\ &\boldsymbol{\nu}x\,\Big(\\
&\quad \overline{x}(q).q(v).\boldsymbol{\nu}x'\,\overline{v}\langle x',p\rangle.x'(r).\overline{x}(q').q'(w).(\overline{r}w \mid !x'(r').\overline{r'}w) \mid\\
&\quad x(r).\boldsymbol{\nu}q'\,(\overline{y}q' \mid\\
&\qquad\qquad q'(w).(\overline{r}w \mid !x(r').\overline{r'}w))\Big)\\
\xrightarrow{\tau}\equiv\ &(\boldsymbol{\nu}x,q,q')\\
&\Big(q(v).\boldsymbol{\nu}x'\,\overline{v}\langle x',p\rangle.x'(r).\overline{x}(q').q'(w).(\overline{r}w \mid !x'(r').\overline{r'}w) \mid\\
&\quad \overline{y}q' \mid\\
&\quad q'(w).(\overline{q}w \mid !x(r').\overline{r'}w)\Big)\\
\xrightarrow{\overline{y}(q')}\xrightarrow{q'(w)}\ &(\boldsymbol{\nu}x,q)\\
&\Big(q(v).\boldsymbol{\nu}x'\,\overline{v}\langle x',p\rangle.x'(r).\overline{x}(q').q'(w).(\overline{r}w \mid !x'(r').\overline{r'}w) \mid\\
&\quad (\overline{q}w \mid !x(r').\overline{r'}w)\Big)\\
\xrightarrow{\tau}\ &(\boldsymbol{\nu}x)\,\Big(\boldsymbol{\nu}x'\,\overline{w}\langle x',p\rangle.x'(r).\overline{x}(q').q'(w).(\overline{r}w \mid !x'(r').\overline{r'}w) \mid\\
&\quad !x(r').\overline{r'}w\Big)\\
\xrightarrow{\boldsymbol{\nu}x'\,\overline{w}\langle x',p\rangle}\xrightarrow{x'(r)}\ &(\boldsymbol{\nu}x)\,\Big(\overline{x}(q').q'(w).(\overline{r}w \mid !x'(r').\overline{r'}w) \mid\\
&\quad !x(r').\overline{r'}w\Big)\\
\xrightarrow{\tau}\xrightarrow{\tau}\equiv\ &\overline{r}w \mid !x'(r').\overline{r'}w
\end{aligned}$$

Again, the above is the only possible trace for the term. Up-to some renaming, the final derivative is the same as the final derivative of the trace emanating from $\mathcal{N}[\![yy]\!]p$ examined earlier. However, the two traces are different—the first is longer. As a consequence, the two terms can be distinguished, for instance in the context

$$C \overset{\text{def}}{=} \boldsymbol{\nu} y\ ([\cdot] \mid y(q).\overline{q}(v).v(x,p).\boldsymbol{\nu} r\ (\overline{x}r \mid y(q').\overline{h}))$$

The observable output at h becomes visible only when the context is filled with the first term, $\mathcal{N}[\![yy]\!]p$ (the one that produces the longer trace).

In contrast, in the call-by-name encoding the validity of the full β-reduction holds [23]. Therefore the above counterexample not only shows that the call-by-need encoding is observably different from the call-by-name one; it also tells us that the properties that the two encoding satisfy are quite different.

4.4 A Refined Encoding

In this section we experiment with a refinement $\mathcal{R}$ of the encoding so to improve on the problems described in Sect. 4.3.

Such encoding is shown in Fig. 3. In the definition of application for call-by-need, the argument of a function is interrogated only once, the first time the argument is used. Future uses of the argument will directly use the answer so received, without repeating the interrogation—this is indeed the essence of the call-by-need optimisation over call-by-name. To mirror this policy we modify the encoding of an abstraction $\lambda x.M$, so that the body M will interrogate the parameter x only once. As a consequence, in this refined encoding when the head 'λx' of the function is consumed a local entry is left that takes care of the dialogue with x; in particular the local entry makes sure that x is consulted only once. The refined encoding, while it exhibits more interactions than the original one, in a distributed setting may be thought of as an optimisation of the latter, as the interactions with the new local entry replace interactions with the possibly remote actual parameter for x. We write $\mathtt{LE}(x, y)$ for a local entry in which the internal resource is x and the external one is y; it will be convenient to break its definition into two parts, using the auxiliary local entry $\mathtt{LE}'(s, r, x)$; see Fig. 3.

The local entry is unnecessary if the internal resource is used only once.

Lemma 4. *Suppose z' appears only once in M. Then*

$$\boldsymbol{\nu} z'\ (\mathcal{R}[\![M]\!]p \mid \mathtt{LE}(z', z)) \approx \mathcal{R}[\![M\{z/z'\}]\!]p$$

Proof. By induction on the structure of M. The most interesting case is when $M = z'$, in which case we exploit the (linear) law of ALπ in Lemma 2. When M is an application we exploit the hypothesis (z' occurring only once), and simple algebraic manipulation, so to be able to carry out the induction.

The next lemma shows that local entries compose.

Lemma 5. $\boldsymbol{\nu} x\ (\mathtt{LE}(z, x) \mid \mathtt{LE}(x, y)) \approx \mathtt{LE}(z, y)$.

Proof. We use laws (2) and (1) of Lemma 1, and the garbage-collection laws of structural congruence.

$$\begin{aligned}
\mathcal{R}[\![\lambda x.\, M]\!]p &\overset{\text{def}}{=} \overline{p}(v).\, !v(x',q).\, \boldsymbol{\nu} x\ (\ \mathcal{R}[\![M]\!]q \\
&\qquad\qquad\qquad\qquad\quad \mid \texttt{LE}(x,x')) \\
\mathcal{R}[\![x]\!]p &\overset{\text{def}}{=} \overline{x}p \\
\mathcal{R}[\![MN]\!]p &\overset{\text{def}}{=} \\
(\boldsymbol{\nu} q\ )\Big(\mathcal{R}[\![M]\!]q \mid q(v).\, &\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.\, x(r).\, \boldsymbol{\nu} q'\ (\mathcal{R}[\![N]\!]q' \mid \\
&\qquad\qquad\qquad\qquad \texttt{LE}'(q',r,x))\Big)
\end{aligned}$$

where

$$\begin{aligned}
\texttt{LE}(x,y) &\overset{\text{def}}{=} x(r).\, \overline{y}(s).\, \texttt{LE}'(s,r,x) \\
\texttt{LE}'(s,r,x) &\overset{\text{def}}{=} s(v).\, (\overline{r}v \mid !x(r').\, \overline{r'}v)
\end{aligned}$$

Fig. 3. The refined call-by-need encoding

We revisite the counterexample of Sect. 4.3, that involves terms yy and $(\lambda z.zz)y$, under the refined encoding $\mathcal{R}$. All free variables should be protected under a local entry, except for the variables that occur only once (by Lemma 4). We begin by examining $\boldsymbol{\nu} y'\ (\mathcal{R}[\![y'y']\!]q \mid \texttt{LE}(y',y))$. We have:

$$\begin{aligned}
& \boldsymbol{\nu} y'\ (\mathcal{R}[\![y'y']\!]q \mid \texttt{LE}(y',y)) \\
\sim\quad & \boldsymbol{\nu} y'\ (\overline{y'}(q).q(v).\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\overline{y'}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \\
& \qquad \mid y'(r).\overline{y}(r').r'(v).(\overline{r}v \mid !y'(r).\overline{r}v)) \\
\approx\quad & (\boldsymbol{\nu} y',q)\ (q(v).\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\overline{y'}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \\
& \qquad\quad \mid \overline{y}(r').r'(v).(\overline{q}v \mid !y'(q).\overline{q}v)) \\
\xrightarrow{\overline{y}(r')}\xrightarrow{r'(v)}\quad & (\boldsymbol{\nu} y',q)\ (q(v).\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\overline{y'}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \\
& \qquad\quad \mid \overline{q}v \mid !y'(q).\overline{q}v) \\
\xrightarrow{\tau}\quad & (\boldsymbol{\nu} y')\ (\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle.x(r).\overline{y'}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \\
& \qquad \mid !y'(q).\overline{q}v) \\
\xrightarrow{\boldsymbol{\nu} x\, \overline{v}\langle x,p\rangle}\xrightarrow{x(r)}\quad & (\boldsymbol{\nu} y')\ (\overline{y'}(q').q'(w).(\overline{r}w \mid !x(r').\overline{r'}w) \\
& \qquad \mid !y'(q).\overline{q}v) \\
\xrightarrow{\tau}\xrightarrow{\tau}\quad & \overline{r}v \mid !x(r').\overline{r'}v
\end{aligned}$$

where the occurrence of $\sim$ is justified by (2) (the two encodings coincide on terms that do not contain abstractions) and definition of $\texttt{LE}(y',y)$, and the occurrence of $\approx$ comes from law (1) of Lemma 1.

We now consider the second term, $(\lambda z.zz)y$, under the refined encoding. First we note that, if the argument of a function L is a variable, then the refined encoding can be simplified thus:

$$\begin{aligned}\mathcal{R}[\![Ly]\!]p = (\boldsymbol{\nu} q\,)\Big(&\mathcal{R}[\![L]\!]q \mid \\ &q(v).\boldsymbol{\nu} x\,\overline{v}\langle x,p\rangle.x(r).\boldsymbol{\nu} q'\,(\mathcal{N}[\![y]\!]q' \mid \\ &\qquad\qquad\qquad\qquad \mathtt{LE}'(q',r,x))\Big)\end{aligned}$$

$$\begin{aligned}= (\boldsymbol{\nu} q\,)\Big(&\mathcal{R}[\![L]\!]q \mid \\ &q(v).\boldsymbol{\nu} x\,\overline{v}\langle x,p\rangle.\mathtt{LE}(x,y)\Big)\end{aligned}$$

Using this property, we have:

$$\begin{aligned}\mathcal{R}[\![(\lambda z.zz)y]\!]p \quad &= \quad (\boldsymbol{\nu} q\,)\Big(\mathcal{R}[\![\lambda z.zz]\!]q \mid \\ &\qquad\qquad q(v).\boldsymbol{\nu} x\,\overline{v}\langle x,p\rangle.\mathtt{LE}(x,y)\Big)\\ &= \quad (\boldsymbol{\nu} q\,)(\overline{q}(v).!v(z',q).\boldsymbol{\nu} z\,(\mathcal{R}[\![zz]\!]q \mid \mathtt{LE}(z,z'))\\ &\qquad\qquad \mid q(v).\boldsymbol{\nu} x\,\overline{v}\langle x,p\rangle.\mathtt{LE}(x,y))\\ &\xrightarrow{\tau}\xrightarrow{\tau} \boldsymbol{\nu} z\,\boldsymbol{\nu} x\,(\mathcal{R}[\![zz]\!]p \mid \mathtt{LE}(z,x) \mid \mathtt{LE}(x,y))\\ &\approx \quad \boldsymbol{\nu} z\,(\mathcal{R}[\![zz]\!]p \mid \mathtt{LE}(z,y))\end{aligned}$$

where $\approx$ is obtained by composition of local entries (Lemma 5).

The above reasoning shows that the behaviour of the initial and refined encodings are the same on the term $(\lambda z.zz)y$:

$$\mathcal{R}[\![(\lambda z.zz)y]\!]p \approx \mathcal{N}[\![(\lambda z.zz)y]\!]p$$

And it is also the same as that of the refined encoding on the β-contractum yy, when the free variables y are protected under the appropriate local entry, i.e.,

$$\mathcal{R}[\![(\lambda zz)y]\!]p \approx \boldsymbol{\nu} y'\,(\mathcal{R}[\![y'y']\!]p \mid \mathtt{LE}(y',y))$$

The result also holds by closing up the terms:

$$\mathcal{R}[\![\lambda y.(yy)]\!]p \approx \mathcal{R}[\![\lambda y.((\lambda z.zz)y)]\!]p$$

In the refined encoding $\mathcal{R}$, the local entry (necessary for confining the behaviour of yy) is produced by the encoding of the abstraction.

More generally, proceeding as above one can show that, on the encoding $\mathcal{R}$, β-reduction is valid when the argument of a function is a variable. Moreover, β-reduction is also valid when the argument is itself a function, reasoning as in Theorem 3. We may therefore conclude that β-reduction is valid when the argument is a value (as it is the case for the encoding of the call-by-value strategy [24]).

We write $\mathtt{LE}(\widetilde{x},\widetilde{y})$ for $\mathtt{LE}(x_1,y_1) \mid \ldots \mid \mathtt{LE}(x_n,y_n)$ where n is the length of the tuples $\widetilde{x}$ and $\widetilde{y}$. We use V to range over values.

Theorem 5. *Suppose* $\mathsf{fv}((\lambda z.MV)) \subseteq \widetilde{x}$. *Then, for any* y *and fresh* $\widetilde{y}$, *we have*

$$\boldsymbol{\nu}\widetilde{x}\,(\mathcal{R}[\![(\lambda z.M)V]\!]p \mid \mathtt{LE}(\widetilde{x},\widetilde{y})) \approx \boldsymbol{\nu}\widetilde{x}\,(\mathcal{R}[\![M\{^{V}\!/\!_{z}\}]\!]p \mid \mathtt{LE}(\widetilde{x},\widetilde{y}))$$

Similarly to what done earlier, a local entry $\mathtt{LE}(x_i, y_i)$ may be removed when the variable x_i appears at most once.

However, even in the encoding $\mathcal{R}$, the full β-reduction is unvalid. As a counterexample we can use the terms $(xz)(xz)$ and $(\lambda y.yy)(xz)$. Indeed we have:

$$\boldsymbol{\nu} xz\, (\mathcal{R}[\![(xz)(xz)]\!]p \mid \mathtt{LE}(x, x') \mid \mathtt{LE}(z, z') \not\approx \mathcal{R}[\![(\lambda y.yy)(xz)]\!]p$$

We omit the calculations, which are rather involved. Intuitively, the difference appears because the full trace of the process computing the application xz is visible twice in the first process, whereas, in the second process, the second time, part of the trace is concealed.

5 Conclusions

Call-by-need was proposed by Wadsworth [28] as an implementation technique. Formalizations of call-by-need on a λ-calculus with a `let` construct or with environments include Ariola et al. [2], Launchbury [11], Purushothaman and Seaman [20], and Yoshida [29]. The uniform encoding in Sect. 4 is from [16]. A study of the correctness of the call-by-need encoding in Fig. 2 is in [5]. Encodings of graph reductions, related to call-by-need, into π-calculus were given in [4,8] but their correctness was not studied. Niehren [15] used encodings of call-by-name, call-by-value and call-by-need λ-calculi into π-calculus to compare the time complexity of the strategies.

In the paper we have used the theory of the Asynchronous Local π-calculus (ALπ) [12] to reason about the encoding of the call-by-need λ-calculus strategy as processes. We have mainly focused on the validity of β-reduction. We have showed that various instances of the property on closed terms hold, though the general property fails. We have also considered a refined encoding in which β-reduction on arbitrary values (though not on arbitrary terms) holds. All this leaves us with some challenging questions, that we leave for future work:

1. In the refined encoding, we use special processes called *local entries* to protect the formal parameter of the function, thus improving the results about β-validity. Is it possible to further protect variables (or terms) so to recover the full β-validity?
2. Is there a different form of behavioural equivalence under which the full β-validity holds, in the initial or in the refined encoding?
3. What is an appropriate process preorder under which call-by-need can indeed be proved to be an optimisation of call-by-name?
4. What is the equivalence on λ-terms induced by the call-by-need encoding? Following the results for call-by-name and call-by-value, one expects to recover some kind of tree structure (Böhm Trees and Lévy-Longo Tree for call-by-name, Lassen's trees [10] for call-by-value). We are not aware of similar tree structures for call-by-need. Hence investigating this question may also shed light on what should the appropriate forms of trees for call-by-need be.

In questions (2) and (3), ‘easy’ answers may be obtained by confining the testing contexts to be encodings of λ-calculus contexts. The challenge is to find more general and useful answers, with applications outside the realm of pure λ-calculi. One may consider forms of behavioural types.

In questions (1) and (2), perhaps requiring the validity of the full β-reduction, in the same way as for call-by-name, is too demanding. Indeed in this way probably the tree structure referred to in question (4) is likely to be the same as that for call-by-name. One may find it acceptable to limit β-validity to reductions between closed terms.

Acknowledgments. Thanks to the reviewers for their careful reading of the paper and their suggestions. Research partly supported by the H2020-MSCA-RISE project ID 778233 “Behavioural Application Program Interfaces (BEHAPI)”.

References

1. Amadio, R., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous π-calculus. Theoret. Comput. Sci. **195**, 291–324 (1998)
2. Ariola, Z., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: A call-by-need λ-calculus. In: Proceedings of the 22th POPL. ACM Press (1995)
3. Boreale, M., Sangiorgi, D.: Some congruence properties for π-calculus bisimilarities. Theoret. Comput. Sci. **198**, 159–176 (1998)
4. Boudol, G.: Some chemical abstract machines. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803, pp. 92–123. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58043-3_18
5. Brock, S., Ostheimer, G.: Process semantics of graph reduction. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 471–485. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60218-6_36
6. Durier, A., Hirschkoff, D., Sangiorgi, D.: Eager functions as processes. In: 33nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018. IEEE Computer Society (2018)
7. Fournet, C., Gonthier, G.: The join calculus: a language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45699-6_6
8. Jeffrey, A.: A chemical abstract machine for graph reduction extended abstract. In: Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 293–303. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58027-1_14
9. Kobayashi, N., Pierce, B., Turner, D.: Linearity and the pi-calculus. TOPLAS **21**(5), 914–947 (1999). Preliminary summary appeared in Proceedings of POPL’96
10. Lassen, S.B.: Eager normal form bisimulation. In: Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), Chicago, IL, USA, 26–29 June 2005, pp. 345–354 (2005)
11. Launchbury, J.: A natural semantics for lazy evaluation. In: Proceedings of the 20th POPL. ACM Press (1993)
12. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. J. Math. Struct. Comput. Sci. **14**(5), 715–767 (2004)

13. Milner, R.: The polyadic π-calculus: a tutorial. Technical report, ECS-LFCS-91-180, LFCS, Department of Computer Science, Edinburgh University, October 1991. Also in Bauer, F.L., Brauer, W., Schwichtenberg, H. (eds.) Logic and Algebra of Specification. Springer, Heidelberg (1991)
14. Milner, R.: Functions as processes. J. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992)
15. Niehren, J.: Functional computation as concurrent computation. In: Proceedings of the 23th POPL. ACM Press (1996)
16. Ostheimer, G., Davie, A.: π-calculus characterisations of some practical λ-calculus reductions strategies. Technical report, CS/93/14, St. Andrews (1993)
17. Palamidessi, C.: Comparing the expressive power of the synchronous and asynchronous pi-calculi. J. Math. Struct. Comput. Sci. **13**(5), 685–719 (2003)
18. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge (2000)
19. Plotkin, G.: Call by name, call by value and the λ-calculus. Theoret. Comput. Sci. **1**, 125–159 (1975)
20. Purushothaman, S., Seaman, J.: An adequate operational semantics of sharing in lazy evaluation. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 435–450. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55253-7_26
21. Sangiorgi, D.: An investigation into functions as processes. In: Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 143–159. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58027-1_7
22. Sangiorgi, D.: From λ to π, or: Rediscovering continuations. J. Math. Struct. Comput. Sci. **9**(4), 367–401 (1999). Special Issue on "Lambda-Calculus and Logic" in Honour of Roger Hindley
23. Sangiorgi, D.: Lazy functions and mobile processes. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge (2000)
24. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
25. Sangiorgi, D.: Typed pi-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. TAPOS **5**(1), 25–33 (1999)
26. Turner, N.: The polymorphic pi-calculus: theory and implementation. Ph.D. thesis, Department of Computer Science, University of Edinburgh (1996)
27. Vasconcelos, V.T.: Lambda and pi calculi, CAM and SECD machines. J. Funct. Program. **15**(1), 101–127 (2005)
28. Wadsworth, C.P.: Semantics and pragmatics of the lambda calculus. Ph.D. thesis, University of Oxford (1971)
29. Yoshida, N.: Optimal reduction in weak lambda-calculus with shared environments. In: Proceedings of the FPCA 1993, Functional Programming and Computer Architecture, pp. 243–252 (1993)