Strong Bisimulation for Control Operators

Delia Kesner
IRIF, Université de Paris and CNRS, France
Institut Universitaire de France (IUF), France

Eduardo Bonelli
Stevens Institute of Technology, Hoboken, NJ, USA

Andrés Viso
Universidad de Buenos Aires, Argentina
Universidad Nacional de Quilmes, Bernal, Buenos Aires, Argentina

Abstract

The purpose of this paper is to identify programs with control operators whose reduction semantics are in exact correspondence. This is achieved by introducing a relation $\simeq$, defined over a revised presentation of Parigot's $\lambda\mu$-calculus we dub ΛM.

Our result builds on two fundamental ingredients: (1) factorization of $\lambda\mu$-reduction into multiplicative and exponential steps by means of explicit term operators of ΛM, and (2) translation of ΛM-terms into Laurent's polarized proof-nets (PPN) such that cut-elimination in PPN simulates our calculus. Our proposed relation $\simeq$ is shown to characterize structural equivalence in PPN. Most notably, $\simeq$ is shown to be a strong bisimulation with respect to reduction in ΛM, i.e. two $\simeq$-equivalent terms have the exact same reduction semantics, a result which fails for Regnier's σ-equivalence in λ-calculus as well as for Laurent's σ-equivalence in $\lambda\mu$.

2012 ACM Subject Classification Theory of computation $\to$ Lambda calculus; Theory of computation $\to$ Linear logic; Theory of computation $\to$ Operational semantics

Keywords and phrases Lambda-mu calculus, proof-nets, strong bisimulation

Digital Object Identifier 10.4230/LIPIcs.CSL.2020.4

Category Invited Talk

Related Version A full version of the paper is available at `https://arxiv.org/abs/1906.09370`.

Funding This work was partially supported by LIA INFINIS, and the ECOS-Sud program PA17C01.

Acknowledgements To Olivier Laurent for fruitful discussions.

1 Introduction

An important topic in the study of programming language theories is unveiling structural similarities between expressions denoting programs. They are widely known as *structural equivalences*; equivalent expressions behaving exactly in the same way. Process calculi are a rich source of examples. In CCS expressions stand for processes in a concurrent system. For example, $P \parallel Q$ denotes the parallel composition of processes P and Q. Structural equivalence includes equations such as the one stating that $P \parallel Q$ and $Q \parallel P$ are equivalent. This minor reshuffling of subexpressions has little impact on the behavior of the overall expression: structural equivalence is a *strong bisimulation* for process reduction. This paper

$$\begin{tikzcd}
o \arrow[d, squiggly] \arrow[r, phantom, "\simeq"] & p \arrow[d, squiggly, dashed] \\
o' \arrow[r, phantom, "\simeq"] & p'
\end{tikzcd}$$

is concerned with such notions of reshuffling of expressions in *λ-calculi with control operators*.

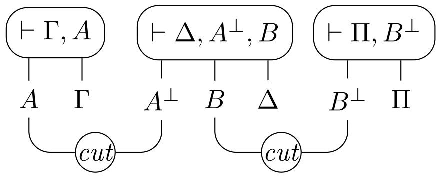
The induced notion of structural equivalence, in the sequel $\simeq$, should identify terms having exactly the same reduction semantics too, that is, should be a strong bisimulation with respect to reduction in these calculi. In other words, $\simeq$ should be symmetric and moreover $o \simeq p$ and $o \rightsquigarrow o'$ should imply the existence of p' such that $p \rightsquigarrow p'$ and $o' \simeq p'$, where $\rightsquigarrow$ denotes some given notion of reduction for control operators (see figure on the right).

Formulating such structural equivalences for the λ-calculus is hindered by the sequential (left-to-right) orientation in which expressions are written. Consider for example the terms $(\lambda x.(\lambda y.t)\, u)\, v$ and $(\lambda x.\lambda y.t)\, v\, u$. They seem to have the same redexes, only permuted, similar to the situation captured by the above mentioned CCS equation. A closer look, however, reveals that this is not entirely correct. The former has two redexes (one indicated below by underlining and another by overlining) and the latter has only one (underlined):

$$\underline{(\lambda x.\overline{(\lambda y.t)\, u})\, v} \text{ and } \underline{(\lambda x.(\lambda y.t))\, v}\, u \tag{1}$$

The overlined redex on the left-hand side is not visible on the right-hand side; it will only reappear, as a newly *created* redex, once the underlined redex is computed. Despite the fact that the syntax gets in the way, Regnier [27] proved that these terms behave in *essentially* the same way. More precisely, he introduced a structural equivalence for λ-terms, known as *σ-equivalence* and he proved that σ-equivalent terms have head, leftmost, perpetual and, more generally, maximal reductions of the same length. However, the mismatch between the terms in (1) is unsatisfying since there clearly seems to be an underlying strong bisimulation, which is not showing itself due to a notational shortcoming. It turns out that through the graphical intuition provided by linear logic *proof-nets*, one can define an enriched λ-calculus that unveils a strong bisimulation for the intuitionistic case [4]. Further details are described below. In this paper, we resort to this same intuition to explore whether it is possible to uncover a strong bisimulation behind a notion of structural equivalence for the more challenging setting of classical logic. Thus, we will not only capture structural equivalence on pure functions, but also on *programs with control operators.* In our case it is polarized proof-nets (PPN) that will serve as semantic yardstick. We next briefly revisit proof-nets and discuss how they help unveil structural equivalence as strong bisimulation for λ-calculi. An explanation of the challenges that we face in addressing the classical case will follow.

Proof-Nets. A *proof-net* is a graph-like structure whose nodes denote logical inferences and whose edges or wires denote the formula they operate on (cf. Sec. 6). Proof-nets were introduced in the setting of linear logic [12], which provides a mechanism to explicitly control the use of resources by restricting the application of the *structural* rules of weakening and contraction. Proof-nets are equipped with an operational semantics specified by graph transformation rules which captures cut elimination in sequent calculus. The resulting cut elimination rules on proof-nets are split into two different kinds: *multiplicative*, that essentially (linearly) reconfigure wires, and *exponential*, which are the only ones that are able to erase or duplicate (sub)proof-nets. The latter are considered to introduce interesting or *meaningful* computation. Most notably, proof-nets abstract away the order in which certain rules occur in a sequent derivation. As an example, assume three derivations of the judgements $\vdash \Gamma, A$, $\vdash \Delta, A^{\perp}, B$ and $\vdash \Lambda, B^{\perp}$, resp. The order in which these derivations are composed via cuts into a single derivation is abstracted away in the resulting proof-net:



In other words, *different* terms/derivations are represented by the *same* proof-net. Hidden structural similarity between terms can thus be studied by translating them to proof-nets. Moreover, following the Curry-Howard isomorphism which relates computation and logic, this correspondence can be extended not only to terms themselves [10, 8, 18, 5] but also to their reduction behavior [2]. In this paper, however, we concentrate on identifying those different classical derivations which translate to the same graph representation. As is standard in the literature, the notion of proof-net identity we adopt includes simple equalities such as *associativity* of contraction nodes and other similar rewirings (cf. notion of *structural equivalence* of proof-nets).

Intuitionistic σ-Equivalence. As mentioned before, Regnier introduced a notion of σ-*equivalence* on λ-terms (written here $\simeq_\sigma$ and depicted in Fig. 1), and proved that σ-equivalent terms behave in essentially identical way. This equivalence relation involves permuting certain redexes, and was unveiled through the study of proof-nets. In particular, following Girard's encoding of intuitionistic into linear logic [12], σ-equivalent terms are mapped to the same proof-net (modulo multiplicative cuts and structural equivalence).

$$\begin{array}{rcll} (\lambda x.\lambda y.t)\,u & \simeq_{\sigma_1} & \lambda y.(\lambda x.t)\,u & y \notin u \\ (\lambda x.t\,v)\,u & \simeq_{\sigma_2} & (\lambda x.t)\,u\,v & x \notin v \end{array}$$

Figure 1 Regnier's σ-equivalence for λ-terms.

The reason why Regnier's result is not immediate is that redexes present on one side of an equation may disappear on the other side of it, as illustrated in the terms in (1). One might rephrase this observation by stating that $\simeq_\sigma$ is *not a strong bisimulation* over the set of λ-terms. If it were, then establishing that σ-equivalent terms behave essentially in the same way would be trivial.

Adopting a more refined view of λ-calculus, as suggested by linear logic, which splits cut elimination on logical derivations into multiplicative and exponential steps yields a decomposition of β-reduction into multiplicative/exponential steps *on terms*. The theory of *explicit substitutions* (a survey can be found in [17]) provides a convenient syntax to reflect these steps at the term level. Indeed, β-reduction can be decomposed into two steps, namely $\mathtt{B}$ (for Beta), which acts *at a distance* [5] in the sense that the abstraction and the argument may be separated by an arbitrary number of explicit substitutions, and $\mathtt{S}$ (for Substitution):

$$\begin{array}{rcl} (\lambda x.t)[x_1\backslash v_1]\ldots[x_n\backslash v_n]\,u & \mapsto_{\mathtt{B}} & t[x\backslash u][x_1\backslash v_1]\ldots[x_n\backslash v_n] \\ t[x\backslash u] & \mapsto_{\mathtt{S}} & t\{x\backslash u\} \end{array} \qquad (2)$$

Firing the $\mathtt{B}$-rule creates an *explicit substitution* operator, written $t[x\backslash u]$, so that $\mathtt{B}$ essentially reconfigures symbols, and indeed reads as a multiplicative cut in proof-nets. The $\mathtt{S}$-rule executes the substitution by performing a replacement of all free occurrences of x in t with u, written $t\{x\backslash u\}$, so that it is $\mathtt{S}$ that performs interesting or *meaningful* computation and reads as an exponential cut in proof-nets.

A term without any occurrence of the left-hand side of rule $\mathtt{B}$ is called a $\mathtt{B}$-normal form; we shall refer to these terms as *canonical forms*. Decomposition of β-reduction by means of the rules in (2) prompts one to replace $\simeq_\sigma$ (Fig. 1) with a new relation $\simeq_{\sigma^B}$ (Fig. 2). The latter is formed essentially by taking the $\mathtt{B}$-*normal form* of each side of the $\simeq_\sigma$ equations[1].

[1] Also included in $\simeq_{\sigma^B}$ is equation $\simeq_{\sigma^B_3}$ allowing commutation of orthogonal (independent) substitutions.

$$\begin{array}{rcll}
(\lambda y.t)[x\backslash u] & \simeq_{\sigma_1^B} & \lambda y.t[x\backslash u] & y \notin u \\
(t\, v)[x\backslash u] & \simeq_{\sigma_2^B} & t[x\backslash u]\, v & x \notin v \\
t[y\backslash v][x\backslash u] & \simeq_{\sigma_3^B} & t[x\backslash u][y\backslash v] & y \notin u, x \notin v
\end{array}$$

Figure 2 Strong bisimulation for λ-terms with explicit substitutions.

Since B-reduction corresponds only to multiplicative cuts in proof-nets, the translation of $\simeq_{\sigma^B}$-equivalent typed terms also yields *structurally equivalent* proof-nets. In other words, $\simeq_{\sigma^B}$-equivalence classes of λ-terms with explicit substitutions in B-normal form are in one-to-one correspondence with intuitionistic linear logic proof-nets [5]. Moreover, $\simeq_{\sigma^B}$ is a strong bisimulation with respect to meaningful reduction (i.e. S-reduction) over the extended set of terms that includes explicit substitutions [5, 4]. Indeed, $\simeq_{\sigma^B}$ is symmetric, and moreover, $u \simeq_{\sigma^B} v$ and $u \to_{\mathtt{S}} u'$ implies the existence of v' such that $v \to_{\mathtt{S}} v'$ and $u' \simeq_{\sigma^B} v'$. Note also that the B-normal form of both sides of (1) are $\simeq_{\sigma^B}$-equivalent, thus repairing the mismatch.

Classical σ-Equivalence. This work sets out to explore structural equivalence for *λ-calculi with control operators*. These calculi include operations to manipulate the context in which a program is executed. We focus here on Parigot's $\lambda\mu$-calculus [25], which extends the λ-calculus with two new operations: $[\alpha]\, t$ (*named term*) and $\mu\alpha.c$ (*μ-abstraction*). The former may be read as "call continuation α with t as argument" and the latter as "record the current continuation as α and continue as c". Reduction in $\lambda\mu$ consists of the β-rule together with:

$$(\mu\alpha.c)\, u \quad \mapsto_\mu \quad \mu\alpha.c\{\!\{\alpha\backslash u\}\!\}$$

where $c\{\!\{\alpha\backslash u\}\!\}$, called here *replacement*, replaces all subexpressions of the form $[\alpha]\, t$ in c with $[\alpha]\,(t\, u)$. Regnier's notion of σ-equivalence for λ-terms was extended to $\lambda\mu$ by Laurent [23] (cf. Fig. 4 in Sec. 4). Here is an example of terms related by this extension, where the redexes are underlined/overlined:

$$(\underline{(\lambda x.\mu\alpha.[\gamma]\, u)\, w})\, v \simeq_\sigma \overline{(\mu\alpha.[\gamma]\, \underline{(\lambda x.u)\, w})\, v}$$

Once again, the fact that a harmless permutation of redexes has taken place is not obvious. The term on the right has two redexes (μ and β) but the one on the left only has one (β) redex. Another, more subtle, example of terms related by Laurent's extension clearly suggests that operational indistinguishability cannot rely on relating arbitrary μ-redexes; the underlined μ-redex on the left does not appear at all on the right:

$$\underline{(\mu\alpha.[\alpha]\, x)\, y} \simeq_\sigma x\, y \tag{3}$$

Clearly, σ-equivalence on $\lambda\mu$-terms *fails to be a strong bisimulation*. Nonetheless, Laurent proved properties for $\simeq_\sigma$ in $\lambda\mu$ similar to those of Regnier for $\simeq_\sigma$ in λ. Again, one has the feeling that there is a strong bisimulation hiding behind σ-equivalence for $\lambda\mu$.

Towards a Strong Bisimulation for Control Operators. We seek to formulate a notion of equivalence for $\lambda\mu$ in the sense that it is concerned with harmless permutation of redexes possibly involving control operators and inducing a strong bisimulation. As per the Curry-Howard isomorphism, proof normalization in classical logic corresponds to computation in

Notice however that the B-expansion of $\simeq_{\sigma_3^B}$-equivalent terms yields $\simeq_\sigma$-equivalent terms again. For example, the B-expansion of $t[y\backslash v][x\backslash u] \simeq_{\sigma_3^B} t[x\backslash u][y\backslash v]$ yields $(\lambda y.(\lambda x.t)\, u)\, v \simeq_{\sigma_1,\sigma_2} (\lambda x.(\lambda y.t)\, v)\, u$.

$$\begin{tikzcd}
\mu\alpha'.([\alpha]\,x)[\![\alpha\backslash_{\alpha'} y]\!] \arrow[r, phantom, "\simeq"] \arrow[d, "\mathtt{R}"'] & x\,y \arrow[d, "\mathtt{R}", "/"{anchor=center, sloped}] \\
\mu\alpha'.[\alpha']\,x\,y \arrow[r, phantom, "\simeq"] & x\,y
\end{tikzcd}$$

Figure 3 Failure of strong bisimulation.

λ-calculi with control operators [13, 25]. Moreover, since classical logic can be translated into *polarized proof-nets* (PPN), as defined by O. Laurent [22, 23], we use PPNs to guide the development in this work. A first step towards our goal involves decomposing the μ-rule as was done for the β-rule with the rules in (2): this produces a rule M (for Mu), to introduce an *explicit replacement*, that also acts at a distance, and another rule R (for Replacement), that executes replacements:

$$\begin{array}{rcl}
(\mu\alpha.c)[x_1\backslash v_1]\ldots[x_n\backslash v_n]\,u & \mapsto_{\mathtt{M}} & (\mu\alpha'.c[\![\alpha\backslash_{\alpha'} u]\!])[x_1\backslash v_1]\ldots[x_n\backslash v_n] \\
c[\![\alpha\backslash_{\alpha'} u]\!] & \mapsto_{\mathtt{R}} & c\{\!\{\alpha\backslash_{\alpha'} u\}\!\}
\end{array} \tag{4}$$

where $c\{\!\{\alpha\backslash_{\alpha'} u\}\!\}$ replaces each sub-expression of the form $[\alpha]\,t$ in c by $[\alpha']\,tu$. Meaningful computation is seen to be performed by R rather than M. This observation is further supported by the fact that both sides of the M-rule translate into the same proof-net (cf. Sec. 6).

Therefore, we tentatively fix our notion of meaningful reduction to be $\mathtt{S} \cup \mathtt{R}$ over the set of canonical forms, the latter now obtained by taking *both* B and M-normal forms. However, in contrast to the intuitionistic case where the decomposition of β into a multiplicative rule B and an exponential rule S suffices for unveiling the strong bisimulation behind Regnier's σ-equivalence in λ-calculus, it turns out that splitting the μ-rule into M and R is not fine-grained enough. There are various examples, that will be developed in this paper, that illustrate that the resulting relation is still not a strong bisimulation. One such example results from taking the BM normal form of the terms in equation (3), as depicted in Fig. 3. This particular use of R on the left seems innocuous. In fact we show that in our proposed calculus and its corresponding translation to PPNs, both terms $\mu\alpha'.([\alpha]\,x)[\![\alpha\backslash_{\alpha'} y]\!]$ and $\mu\alpha'.[\alpha']\,x\,y$ denote structurally equivalent PPNs (cf. Sec. 6 for a detailed discussion). In any case, this example prompts us to further inquire on the fine structure of R. In particular, we will argue (Sec. 4) that rule R should be further decomposed into *several* independent notions, each one behaving differently with respect to PPNs, and thus with respect to our strong bisimulation $\simeq$. Identifying these notions and their interplay in order to expose the strong bisimulation hidden behind Laurent's σ-equivalence is the challenge we address in this work.

Contributions. The multiplicative/exponential splitting of the intuitionistic case applied to the classical case, falls noticeably short in identifying programs with control operators whose reduction semantics are in exact correspondence. The need to further decompose rule R is rather unexpected, and our proposed decomposition turns out to be subtle yet admits a natural translation to PPN. Moreover, it allows us to obtain a novel and far from obvious strong bisimulation result, highlighting the deep correspondence between PPNs and classical term calculi. Our contributions may be summarized as follows:

1. A refinement of $\lambda\mu$, called ΛM-calculus, including explicit substitutions for variables (resp. explicit replacement for names), and being confluent (Thm. 5).
2. A natural interpretation of ΛM into PPN. More precisely, ΛM-reduction can be implemented by PPN cut elimination (Thm. 14).

3. A notion of structural equivalence $\simeq$ for ΛM which:
   a. characterizes PPN modulo structural equivalence (Thm. 21);
   b. is conservative over Laurent's original equivalence $\simeq_\sigma$ (Thm. 22);
   c. is a strong bisimulation with respect to meaningful steps (Thm. 25).

Structure of the Paper. Sec. 2 and 3 present $\lambda\mu$ and ΛM, resp. Sec. 4 presents a further refinement of ΛM. Sec. 5 defines typed ΛM-objects, Sec. 6 defines polarized proof-nets, and presents the translation from the former to the latter. Sec. 7 presents our equivalence $\simeq$. Its properties are discussed and proved in Sec. 8 and Sec. 9. Finally, Sec. 10 concludes and describes related work. Most proofs can be found in [7] including extended details on PPNs, whose presentation has been abridged in this paper.

2 The $\lambda\mu$-calculus

Preliminary Concepts. A rewrite system $\mathcal{R}$ is a set of objects and a binary (one-step) *reduction* relation $\rightarrow_{\mathcal{R}}$ over those objects. We write $\twoheadrightarrow_{\mathcal{R}}$ (resp. $\rightarrow^{+}_{\mathcal{R}}$) for the reflexive-transitive (resp. transitive) closure of $\rightarrow_{\mathcal{R}}$. A term t is in $\mathcal{R}$-normal form, written $t \in \mathcal{R}$-nf or simply $t \in \mathcal{R}$, if there is no t' s.t. $t \rightarrow_{\mathcal{R}} t'$.

Syntax. We fix a countable infinite set of **variables** $x, y, z, \ldots$ and **continuation names** $\alpha, \beta, \gamma, \ldots$. The set of **objects** $\mathcal{O}(\lambda\mu)$, **terms** $\mathcal{T}(\lambda\mu)$, **commands** $\mathcal{C}(\lambda\mu)$ and **contexts** of the $\lambda\mu$-calculus are given by the following grammar:

Objects	o	$::=$	$t \mid c$
Terms	t	$::=$	$x \mid t\,t \mid \lambda x.t \mid \mu\alpha.c$
Commands	c	$::=$	$[\alpha]\,t$
Contexts	$\mathtt{O}$	$::=$	$\mathtt{T} \mid \mathtt{C}$
Term Context	$\mathtt{T}$	$::=$	$\Box \mid \mathtt{T}\,t \mid t\,\mathtt{T} \mid \lambda x.\mathtt{T} \mid \mu\alpha.\mathtt{C}$
Command Context	$\mathtt{C}$	$::=$	$\boxdot \mid [\alpha]\,\mathtt{T}$

The term $(\ldots((t\,u_1)\,u_2)\ldots)\,u_n$ abbreviates as $t\,u_1\,u_2\ldots u_n$. The grammar extends λ-terms with two new constructors: commands $[\alpha]\,t$ and μ-abstractions $\mu\alpha.c$. Regarding contexts, there are two holes $\Box$ and $\boxdot$ of sort term ($\mathtt{t}$) and command ($\mathtt{c}$) respectively. We write $\mathtt{O}\langle o\rangle$ to denote the replacement of the hole $\Box$ (resp. $\boxdot$) by a term (resp. by a command). We often decorate contexts or functions over expressions with sorts $\mathtt{t}$ and $\mathtt{c}$. For example, $\mathtt{O}_{\mathtt{t}}$ is a context $\mathtt{O}$ with a hole of sort *term*. The subscript is omitted if it is clear from the context.

Free and **bound variables** of objects are defined as expected, in particular $\mathtt{fv}(\mu\alpha.c) \overset{def}{=} \mathtt{fv}(c)$ and $\mathtt{fv}([\alpha]\,t) \overset{def}{=} \mathtt{fv}(t)$. **Free names** of objects are defined as follows: $\mathtt{fn}(x) \overset{def}{=} \emptyset$, $\mathtt{fn}(\lambda x.t) \overset{def}{=} \mathtt{fn}(t)$, $\mathtt{fn}(t\,u) \overset{def}{=} \mathtt{fn}(t) \cup \mathtt{fn}(u)$, $\mathtt{fn}(\mu\alpha.c) \overset{def}{=} \mathtt{fn}(c) \setminus \{\alpha\}$, and $\mathtt{fn}([\alpha]\,t) \overset{def}{=} \mathtt{fn}(t) \cup \{\alpha\}$. **Bound names** are defined accordingly. We use $\mathtt{fv}_x(o)$ (resp. $\mathtt{fn}_\alpha(o)$) to denote the number of free occurrences of the variable x (resp. name α) in o. We write $x \notin o$ (resp. $\alpha \notin o$) if $x \notin \mathtt{fv}(o)$ and $x \notin \mathtt{bv}(o)$ (resp. $\alpha \notin \mathtt{fn}(o)$ and $\alpha \notin \mathtt{bn}(o)$). This notion is extended to contexts as expected.

We work with the standard notion of **α-conversion** i.e. renaming of bound variables and names, thus for example $[\delta]\,(\mu\alpha.[\alpha]\,(\lambda x.x))\,z \equiv_\alpha [\delta]\,(\mu\beta.[\beta]\,(\lambda y.y))\,z$. In particular, when using two different symbols to denote bound variables (resp. names), we assume that they are distinct, without explicitly mentioning it.

Semantics. **Application** of the **substitution** $\{x\backslash u\}$ to the object o, written $o\{x\backslash u\}$, may require α-conversion in order to avoid capture of free variables/names, and it is defined as expected. **Application** of the **replacement** $\{\!\{\alpha\backslash_{\alpha'} u\}\!\}$ to an object o, where $\alpha \neq \alpha'$, written $o\{\!\{\alpha\backslash_{\alpha'} u\}\!\}$, passes the term u as an argument to any sub-command of o of the form $[\alpha]\, t$ and changes the name of α to α'. This operation is also defined modulo α-conversion in order to avoid the capture of free variables/names. Formally:

$$\begin{array}{rcll}
x\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & x & \\
(t\, v)\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & t\{\!\{\alpha\backslash_{\alpha'} u\}\!\}\, v\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \\
(\lambda x.t)\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & \lambda x.t\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & x \notin u \\
(\mu\beta.c)\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & \mu\beta.c\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \beta \notin (u, \alpha, \alpha') \\
([\alpha]\, c)\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & [\alpha']\, (c\{\!\{\alpha\backslash_{\alpha'} u\}\!\}\, u) & \\
([\beta]\, c)\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \overset{def}{=} & [\beta]\, c\{\!\{\alpha\backslash_{\alpha'} u\}\!\} & \beta \neq \alpha
\end{array}$$

For example, if $\mathtt{I} = \lambda z.z$, then $((\mu\alpha.[\alpha]\, x)\, (\lambda z.z\, x))\{x\backslash \mathtt{I}\}$ is equal to $(\mu\alpha.[\alpha]\, \mathtt{I})\, (\lambda z.z\, \mathtt{I})$, and $([\alpha]\, x\, (\mu\beta.[\alpha]\, y))\{\!\{\alpha\backslash_{\alpha'} \mathtt{I}\}\!\} = [\alpha']\, x\, (\mu\beta.[\alpha']\, y\, \mathtt{I})\, \mathtt{I}$.

▶ **Definition 1.** *The $\lambda\mu$-calculus is given by the set $\mathcal{O}(\lambda\mu)$ and the $\lambda\mu$-**reduction relation** $\rightarrow_{\lambda\mu}$, defined as the closure by all contexts of the following rewriting rules*[2] *(equivalently, $\rightarrow_{\lambda\mu} \overset{def}{=} \mathtt{O_t}\langle \mapsto_\beta \cup \mapsto_\mu \rangle$):*

$$\begin{array}{lll}
(\lambda x.t)\, u & \mapsto_\beta & t\{x\backslash u\} \\
(\mu\alpha.c)\, u & \mapsto_\mu & \mu\alpha'.c\{\!\{\alpha\backslash_{\alpha'} u\}\!\}
\end{array}$$

Various control operators can be expressed in the $\lambda\mu$-calculus [11, 21]. A typical example of expressiveness of the $\lambda\mu$-calculus is the control operator **call-cc** [13], specified by the term $\lambda x.\mu\alpha.[\alpha]\, x\, (\lambda y.\mu\delta.[\alpha]\, y)$. The term **call-cc** is assigned the type $((A \rightarrow B) \rightarrow A) \rightarrow A$ (Peirce's Law) in the simply typed $\lambda\mu$-calculus, thus capturing classical logic.

The Notion of σ-Equivalence for $\lambda\mu$-Terms. As in λ-calculus, structural equivalence for $\lambda\mu$ captures inessential permutation of redexes, but this time also involving the control constructs. Laurent's notion of σ-equivalence for $\lambda\mu$-terms [23] (written here also $\simeq_\sigma$) is depicted in Fig. 4. The first two equations are exactly those of Regnier (hence $\simeq_\sigma$ on $\lambda\mu$-terms strictly extends $\simeq_\sigma$ on λ-terms); the remaining ones involve interactions between control operators themselves or control operators and application and abstraction.

Laurent proved properties for $\simeq_\sigma$ similar to those of Regnier for $\simeq_\sigma$. More precisely, $u \simeq_\sigma v$ implies that u is normalizable (resp. is head normalizable, strongly normalizable) iff v is normalizable (resp. is head normalizable, strongly normalizable) [23, Prop. 35]. Based on Girard's encoding of classical into linear logic [12], he also proved that the translation of the left and right-hand sides of the equations of $\simeq_\sigma$, in a typed setting, yield structurally equivalent PPNs [23, Thm. 41]. These results are non-trivial because the left and right-hand side of the equations in Fig. 4 do not have the same β and μ redexes. For example, $(\mu\alpha.[\alpha]\, x)\, y$ and $x\, y$ are related by equation σ_8, however the former has a μ-redex (more precisely it has a *linear* μ-redex) and the latter has none. Indeed, as mentioned in Sec. 1 (cf. the terms in (3)), $\simeq_\sigma$ is not a strong bisimulation with respect to $\lambda\mu$-reduction (cf. Fig. 5). There are other

[2] Parigot [25]'s μ-rule $(\mu\alpha.c)\, u \mapsto_\mu \mu\alpha.c\{\!\{\alpha\backslash u\}\!\}$, relies on a binary replacement operation $\{\!\{\alpha\backslash u\}\!\}$ assigning $[\alpha]\, (t\{\!\{\alpha\backslash u\}\!\})\, u$ to $[\alpha]\, t$ (thus not changing the name of the command). We remark that $\mu\alpha.c\{\!\{\alpha\backslash u\}\!\} \equiv_\alpha \mu\alpha'.c\{\!\{\alpha\backslash_{\alpha'} u\}\!\}$. We adopt here the ternary presentation of the replacement operator [20], because it naturally extends to that of the ΛM-calculus in Sec. 3.

$$\begin{array}{rcll}
(\lambda y.\lambda x.t)\,v & \simeq_{\sigma_1} & \lambda x.(\lambda y.t)\,v & x \notin v \\
(\lambda x.t\,v)\,u & \simeq_{\sigma_2} & (\lambda x.t)\,u\,v & x \notin v \\
(\lambda x.\mu\alpha.[\beta]\,u)\,w & \simeq_{\sigma_3} & \mu\alpha.[\beta]\,(\lambda x.u)\,w & \alpha \notin w \\
[\alpha']\,(\mu\alpha.[\beta']\,(\mu\beta.c)\,w)\,v & \simeq_{\sigma_4} & [\beta']\,(\mu\beta.[\alpha']\,(\mu\alpha.c)\,v)\,w & \alpha \notin w, \beta \notin v, \beta \neq \alpha', \alpha \neq \beta' \\
[\alpha']\,(\mu\alpha.[\beta']\,\lambda x.\mu\beta.c)\,v & \simeq_{\sigma_5} & [\beta']\,\lambda x.\mu\beta.[\alpha']\,(\mu\alpha.c)\,v & x \notin v, \beta \notin v, \beta \neq \alpha', \alpha \neq \beta' \\
[\alpha']\,\lambda x.\mu\alpha.[\beta']\,\lambda y.\mu\beta.c & \simeq_{\sigma_6} & [\beta']\,\lambda y.\mu\beta.[\alpha']\,\lambda x.\mu\alpha.c & \beta \neq \alpha', \alpha \neq \beta' \\
[\alpha]\,\mu\beta.c & \simeq_{\sigma_7} & c\{\beta\backslash\alpha\} & \\
\mu\alpha.[\alpha]\,v & \simeq_{\sigma_8} & v & \alpha \notin v
\end{array}$$

Figure 4 Laurent's σ-equivalence for $\lambda\mu$-terms.

$$\begin{tikzcd}
(\mu\alpha.[\alpha]\,x)\,y \arrow[r, phantom, "\simeq_{\sigma_8}"] \arrow[d, "\mu"'] & x\,y \arrow[d, "/" marking, "\mu"] \\
\mu\alpha.[\alpha]\,x\,y \arrow[r, phantom, "\simeq_{\sigma_8}"] & x\,y
\end{tikzcd}$$

Figure 5 Laurent's $\simeq_\sigma$ equivalence not a strong bisimulation.

examples illustrating that $\simeq_\sigma$ is not a strong bisimulation (cf. Sec. 7). It seems natural to wonder whether, just like in the intuitionistic case, a more refined notion of $\lambda\mu$-reduction could change this state of affairs; that is a challenge we take up in this paper.

3 The ΛM-calculus

We now extend the syntax of $\lambda\mu$ to that of ΛM. We again fix a countable infinite set of **variables** $x, y, z, \ldots$ and **continuation names** $\alpha, \beta, \gamma, \ldots$. The set of **objects** $\mathcal{O}(\Lambda M)$, **terms** $\mathcal{T}(\Lambda M)$, **commands** $\mathcal{C}(\Lambda M)$, **stacks** and **contexts** are given by the following grammar:

Objects	o	$::=$	$t \mid c \mid s$
Terms	t	$::=$	$x \mid t\,t \mid \lambda x.t \mid \mu\alpha.c \mid t[x\backslash t]$
Commands	c	$::=$	$[\alpha]\,t \mid c[\![\alpha\backslash_{\alpha'} s]\!]$
Stacks	s	$::=$	$\# \mid t \cdot s$
Contexts	$\mathtt{O}$	$::=$	$\mathtt{T} \mid \mathtt{C} \mid \mathtt{S}$
Term Contexts	$\mathtt{T}$	$::=$	$\Box \mid \mathtt{T}\,t \mid t\,\mathtt{T} \mid \lambda x.\mathtt{T} \mid \mu\alpha.\mathtt{C} \mid \mathtt{T}[x\backslash t] \mid t[x\backslash \mathtt{T}]$
Command Contexts	$\mathtt{C}$	$::=$	$\boxdot \mid [\alpha]\,\mathtt{T} \mid \mathtt{C}[\![\alpha\backslash_{\alpha'} s]\!] \mid c[\![\alpha\backslash_{\alpha'} \mathtt{S}]\!]$
Stack Contexts	$\mathtt{S}$	$::=$	$\mathtt{T} \cdot s \mid t \cdot \mathtt{S}$
Substitution Contexts	$\mathtt{L}$	$::=$	$\Box \mid \mathtt{L}[x\backslash t]$
Replacement Contexts	$\mathtt{R}$	$::=$	$\boxdot \mid \mathtt{R}[\![\alpha\backslash_{\alpha'} s]\!]$

Terms of $\lambda\mu$ are enriched with **explicit substitutions** of the form $t[x\backslash u]$. Commands of $\lambda\mu$ are enriched with **explicit replacements** of the form $c[\![\alpha\backslash_{\alpha'} s]\!]$, where $\alpha \neq \alpha'$, and α' is called a **replacement name**. Stacks are empty ($\#$) or non-empty ($t \cdot s$). Explicit replacements with empty stacks (i.e. $[\![\beta\backslash_\alpha \#]\!]$) are called **renaming replacements**, otherwise **stack replacements**.

Stack concatenation, denoted $s \cdot s'$, is defined as expected, where $\_ \cdot \_$ is associative and $\#$ is the neutral element. We often write $t_1 \cdot \ldots \cdot t_n \cdot \#$ simply as $t_1 \cdot \ldots \cdot t_n$ (thus, in particular, $u \cdot \#$ is abbreviated as u). Moreover, given a term u, we use the abbreviation $u :: s$

for the term u if $s = \#$ and $((u\,t_1)\ldots)t_n$ if $s = t_1 \cdot \ldots \cdot t_n$. This operation is left-associative, hence $u :: s_1 :: s_2$ means $(u :: s_1) :: s_2$.

Free and **bound variables** of ΛM-objects are extended as expected. In particular, $\mathtt{fv}(t[x\backslash u]) \overset{def}{=} \mathtt{fv}(t) \setminus \{x\} \cup \mathtt{fv}(u)$, and $\mathtt{fv}(c\llbracket\gamma\backslash_{\gamma'} s\rrbracket) \overset{def}{=} \mathtt{fv}(c) \cup \mathtt{fv}(s)$, while $\mathtt{fn}(t[x\backslash u]) \overset{def}{=} \mathtt{fv}(t) \cup \mathtt{fv}(u)$, and $\mathtt{fn}(c\llbracket\gamma\backslash_{\gamma'} s\rrbracket) \overset{def}{=} \mathtt{fn}(c) \setminus \{\gamma\} \cup \{\gamma'\} \cup \mathtt{fn}(s)$. We work, as usual, modulo α-conversion so that bound variables and names can be renamed. Thus e.g. $x[x\backslash u] \equiv_\alpha y[y\backslash u]$ and $\mu\gamma.[\gamma]\,x \equiv_\alpha \mu\beta.[\beta]\,x$. In particular, we will always assume by α-conversion, that $x \notin \mathtt{fv}(u)$ in the term $t[x\backslash u]$ and $\alpha \notin \mathtt{fn}(s)$ in the command $c\llbracket\alpha\backslash_{\alpha'} s\rrbracket$.

The notions of free and bound variables and names are extended to contexts by defining $\mathtt{fv}(\Box) = \mathtt{fv}(\boxdot) = \mathtt{fn}(\Box) = \mathtt{fn}(\boxdot) = \emptyset$. A variable x **occurs bound** in $\mathtt{O}$ if, for any fresh variable $y \neq x$, it occurs in $\mathtt{O}\langle y\rangle$ but not free. Similarly for names. Thus for example x is bound in $\lambda x.\Box$ and $(\lambda x.x)\,\Box$ and α is bound in $\boxdot\llbracket\alpha\backslash_{\alpha'} s\rrbracket$. We use $\mathtt{fv}(o_1, o_2)$ (resp. $\mathtt{fn}(o_1, o_2)$) to abbreviate $\mathtt{fv}(o_1) \cup \mathtt{fv}(o_2)$. (resp. $\mathtt{fn}(o_1) \cup \mathtt{fn}(o_2)$) and also $\mathtt{fn}(o, \alpha)$ to abbreviate $\mathtt{fn}(o) \cup \{\alpha\}$. An object o is **free for a context** $\mathtt{O}$, written $\mathtt{fc}(o, \mathtt{O})$, if the bound variables and bound names of $\mathtt{O}$ do not occur free in o. Thus for example $\mathtt{fc}(zy, \lambda x.(\Box[x'\backslash w]))$ holds but $\mathtt{fc}(xy, \lambda x.\Box)$ does not hold. This notation is naturally extended to sets, i.e. $\mathtt{fc}(S, \mathtt{O})$ means that the bound variables and bound names of $\mathtt{O}$ do not occur free in any element of S. We write $x \notin o$ (resp. $\alpha \notin o$) if $x \notin \mathtt{fv}(o)$ and $x \notin \mathtt{bv}(o)$ (resp. $\alpha \notin \mathtt{fn}(o)$ and $\alpha \notin \mathtt{bn}(o)$). This notion is extended to contexts as expected.

As in Sec. 2, we use $o\{x\backslash u\}$ and $o\{\!\{\alpha\backslash_{\alpha'} s\}\!\}$ to denote, respectively, the natural extensions of the **substitution** and **replacement** operations to ΛM-objects. Both are defined modulo α-conversion to avoid capture of free variables/names. While the first notion is standard, we formalise the second one.

▶ **Definition 2.** *Given $\alpha \notin \mathtt{fn}(s)$, the replacement $o\{\!\{\alpha\backslash_{\alpha'} s\}\!\}$ is defined as follows:*

$$\begin{array}{rcll}
x\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & x & \\
(t\,u)\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & t\{\!\{\alpha\backslash_{\alpha'} s\}\!\}\,u\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \\
(\lambda x.t)\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & \lambda x.t\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & x \notin s \\
(\mu\beta.c)\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & \mu\beta.c\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \beta \notin (s, \alpha, \alpha') \\
t[x\backslash u]\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & t\{\!\{\alpha\backslash_{\alpha'} s\}\!\}[x\backslash u\{\!\{\alpha\backslash_{\alpha'} s\}\!\}] & x \notin s \\
([\alpha]\,t)\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & [\alpha']\,(t\{\!\{\alpha\backslash_{\alpha'} s\}\!\} :: s) & \\
([\beta]\,t)\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & [\beta]\,t\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \alpha \neq \beta \\
c\llbracket\gamma\backslash_\beta s'\rrbracket\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & c\{\!\{\alpha\backslash_{\alpha'} s\}\!\}\llbracket\gamma\backslash_\beta s'\{\!\{\alpha\backslash_{\alpha'} s\}\!\}\rrbracket & \alpha \neq \beta \\
c\llbracket\gamma\backslash_\alpha \#\rrbracket\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & c\{\!\{\alpha\backslash_{\alpha'} s\}\!\}\llbracket\gamma\backslash_\beta s\rrbracket\llbracket\beta\backslash_{\alpha'} \#\rrbracket & s \neq \#, \beta \textit{ fresh} \\
c\llbracket\gamma\backslash_\alpha \#\rrbracket\{\!\{\alpha\backslash_{\alpha'} \#\}\!\} & \overset{def}{=} & c\{\!\{\alpha\backslash_{\alpha'} \#\}\!\}\llbracket\gamma\backslash_{\alpha'} \#\rrbracket & \\
c\llbracket\gamma\backslash_\alpha s'\rrbracket\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & c\{\!\{\alpha\backslash_{\alpha'} s\}\!\}\llbracket\gamma\backslash_{\alpha'} s'\{\!\{\alpha\backslash_{\alpha'} s\}\!\} \cdot s\rrbracket & s' \neq \# \\
\#\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & \# & \\
(t \cdot s')\{\!\{\alpha\backslash_{\alpha'} s\}\!\} & \overset{def}{=} & t\{\!\{\alpha\backslash_{\alpha'} s\}\!\} \cdot s'\{\!\{\alpha\backslash_{\alpha'} s\}\!\} &
\end{array}$$

E.g. $([\alpha]\,x)\{\!\{\alpha\backslash_\gamma y_1 \cdot y_2\}\!\} = [\gamma]\,x\,y_1\,y_2$, and $([\alpha]\,x)\llbracket\beta\backslash_\alpha z_1\rrbracket\{\!\{\alpha\backslash_\gamma y_1\}\!\} = ([\gamma]\,x\,y_1)\llbracket\beta\backslash_\gamma z_1 \cdot y_1\rrbracket$, while $([\alpha]\,x)\llbracket\beta\backslash_\alpha \#\rrbracket\{\!\{\alpha\backslash_\gamma y_1 \cdot y_2\}\!\} = ([\gamma]\,x\,y_1\,y_2)\llbracket\beta\backslash_{\gamma'} y_1 \cdot y_2\rrbracket\llbracket\gamma'\backslash_\gamma \#\rrbracket$.

When $s = \#$, the replacement operation $\_\{\!\{\alpha\backslash_{\alpha'} s\}\!\}$ is called a **renaming**. Most of the cases in the definition above are straightforward, we only comment on the interesting ones. When the (meta-level) replacement operator affects a renaming replacement, i.e. in the case $c\llbracket\gamma\backslash_\alpha \#\rrbracket\{\!\{\alpha\backslash_{\alpha'} s\}\!\}$, the renaming $\llbracket\gamma\backslash_\alpha \#\rrbracket$ is *blocking* the replacement, so that an explicit replacement $\llbracket\gamma\backslash_\beta s\rrbracket$ with a fresh name β is created, and β is then renamed to α'. Regarding the last clause of the definition for commands, since the explicit replacement $\llbracket\gamma\backslash_\alpha s'\rrbracket$ is blocking, it accumulates any additional arguments for γ, hence why *stacks* (i.e. sequences of terms) are used, instead of terms, as target for names.

The two operations $o\{x\backslash u\}$, and $o\{\!\{\alpha\backslash_{\alpha'} s\}\!\}$ are extended to contexts as expected.

▶ **Definition 3.** *The* ΛM***-calculus*** *is given by the set of objects* $\mathcal{O}(\Lambda M)$ *and the* ΛM***-reduction relation*** $\rightarrow_{\Lambda M}$*, defined as the closure by all contexts of the rewriting rules:*

$$\begin{array}{rclcrcl} \mathtt{L}\langle\lambda x.t\rangle\, u & \mapsto_{\mathtt{B}} & \mathtt{L}\langle t[x\backslash u]\rangle & \qquad & \mathtt{L}\langle\mu\alpha.c\rangle\, u & \mapsto_{\mathtt{M}} & \mathtt{L}\langle\mu\alpha'.c[\![\alpha\backslash_{\alpha'} u\cdot\#]\!]\rangle \\ t[x\backslash u] & \mapsto_{\mathtt{S}} & t\{x\backslash u\} & & c[\![\alpha\backslash_{\alpha'} s]\!] & \mapsto_{\mathtt{R}} & c\{\!\{\alpha\backslash_{\alpha'} s\}\!\} \end{array}$$

where $\mapsto_{\mathtt{B}}$ *and* $\mapsto_{\mathtt{M}}$ *are both constrained by the condition* $\mathtt{fc}(u,\mathtt{L})$*, and* $\mapsto_{\mathtt{M}}$ *also requires* $\alpha' \notin (c,u,\alpha,\mathtt{L})$*, thus both rules pull the list context* $\mathtt{L}$ *out by avoiding the capture of free variables/names of* u*. Equivalently,* $\rightarrow_{\Lambda M} \overset{def}{=} \mathtt{O_t}\langle\mapsto_{\mathtt{B}} \cup \mapsto_{\mathtt{S}} \cup \mapsto_{\mathtt{M}}\rangle \cup \mathtt{O_c}\langle\mapsto_{\mathtt{R}}\rangle$*. Given* $X \in \{\mathtt{B},\mathtt{S},\mathtt{M},\mathtt{R}\}$*, we write* $\rightarrow_X$ *for the closure by all contexts of* $\mapsto_X$*.*

Note that $\mathtt{B}$ and $\mathtt{M}$ above, also presented in (2) and (4) of the introduction, operate *at a distance* [5], a characteristic in line with our semantical development being guided by Proof-Nets. Also, following Parigot [25], one might be tempted to rephrase the reduct of $\mathtt{M}$ with a binary constructor, writing $\mathtt{L}\langle\mu\alpha.c[\![\alpha\backslash u]\!]\rangle$. But this is imprecise since free occurrences of α in c cannot be bound to both $\mu\alpha$ and $[\![\alpha\backslash u]\!]$. The subscript α' in $c[\![\alpha\backslash_{\alpha'} u]\!]$ shall replace α in c as described above. The ΛM-calculus implements the $\lambda\mu$-calculus by means of more atomic steps, i.e.

▶ **Lemma 4.** *Let* $o \in \mathcal{O}(\lambda\mu)$*. If* $o \rightarrow_{\lambda\mu} o'$*, then* $o \twoheadrightarrow_{\Lambda M} o'$*.*

Just like $\lambda\mu$, the ΛM-calculus is confluent too. This is proved by using the interpretation method [15], where ΛM is interpreted into $\lambda\mu$ by means of a suitable projection function.

▶ **Theorem 5.** *The* $\rightarrow_{\Lambda M}$ *relation is confluent.*

Proof. By the interpretation method [15], using confluence of $\rightarrow_{\lambda\mu}$ [25]. Details in [7]. ◀

4 Refining Replacement

Now that we have introduced the relation ΛM and hence the reader has a clearer picture of the presentation/implementation of $\lambda\mu$ we will be working with, we briefly revisit our objective. We seek to identify a relation $\simeq$ on a *subset* of ΛM-terms (which we call *canonical*) that is a strong bisimulation for a *subset* of ΛM-reduction (which we call *meaningful*). The proof-net translation of the intuitionistic case suggests that $\simeq$ be defined on the subset of ΛM terms that are in $\mathtt{BM}$-normal form, since a term and its $\mathtt{BM}$-reduct are essentially different syntactic presentations of the same thing. Consequently, we can in principle declare $\mathtt{S} \cup \mathtt{R}$ as the subset of ΛM-reduction that is *meaningful*. However the proof-net translation of ΛM suggests that meaningful reduction for ΛM is the exponential one, manipulating *boxes* (cf. Sec. 6), which allow in particular their erasure and duplication, and unfortunately $\mathtt{R}$ also includes cases without any box manipulation. Moreover, as hinted at in the introduction, when incorporating the $\mathtt{BM}$-normal form of Laurent's σ-equivalence equations into $\simeq$, one immediately realizes that strong bisimulation fails. The heart of the matter is that $\mathtt{R}$ is too course-grained and that we should break it down, weeding out those instances that present an obstacle to strong bisimulation. Indeed, one can distinguish between linear and non-linear instances of $\mathtt{R}$, the translation of the latter involving boxes while the former's not. This section presents a refinement of $\mathtt{R}$, in four stages, identifying a subset of replacement $\mathtt{R}$ we dub *meaningful replacement reduction*. The latter, together with $\mathtt{S}$, will conform the whole notion of *meaningful reduction* (Def 9).

$$\begin{tikzcd}
([\alpha]\,\mu\beta.c)\llbracket\alpha\backslash_{\alpha'}s\rrbracket \arrow[d, "\mathtt{R}"'] & \simeq_{\sigma_7} & c\{\beta\backslash\alpha\}\llbracket\alpha\backslash_{\alpha'}s\rrbracket \arrow[dd, "\mathtt{R}"'] \\
{[\alpha']\,(\mu\beta.c\{\!\{\alpha\backslash_{\alpha'}s\}\!\})\,s} \arrow[d, "\mathsf{BM}"'] & & \\
{[\alpha']\,\mu\beta'.\mathsf{BM}(c\{\!\{\alpha\backslash_{\alpha'}s\}\!\}\llbracket\beta\backslash_{\beta'}s\rrbracket)} & ??? & \mathsf{BM}(c\{\beta\backslash\alpha\}\{\!\{\alpha\backslash_{\alpha'}s\}\!\})
\end{tikzcd}$$

Figure 6 Implicit renaming and strong bisimulation.

Stage 1: Renaming vs Stack Replacement. In this first stage we split $\mathtt{R}$ according to the nature of the explicit replacement, renaming or stack:

$$\begin{array}{rcll}
c\llbracket\alpha\backslash_{\alpha'}\#\rrbracket & \mapsto_{\mathtt{R}_{\#}} & c\{\!\{\alpha\backslash_{\alpha'}\#\}\!\} & \\
c\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \mapsto_{\mathtt{R}_{\neg\#}} & c\{\!\{\alpha\backslash_{\alpha'}s\}\!\} & \text{if } s \neq \#
\end{array}$$

Accordingly, we call $\mathtt{R}_{\#}$ the renaming replacement rule and $\mathtt{R}_{\neg\#}$ the stack replacement rule. The renaming replacement rule unfortunately throws away important information that is required for our strong-bisimulation. As an example, consider the equation σ_7 of Fig. 4, but under a ΛM context containing an explicit replacement $\llbracket\alpha'\backslash_{\alpha}s\rrbracket$, as depicted in Fig. 6. Firing $\llbracket\alpha'\backslash_{\alpha}s\rrbracket$ on the left leads to the creation of another stack replacement redex on the left, with no such redex mimicking it on the right. Indeed, the term on the lower left hand corner has a pending explicit replacement and hence cannot be equated with the term on the lower right hand corner.

As for our notion of *meaningful replacement* reduction, this leads us to disregard the renaming replacement rule, leaving renaming replacements in terms as is, that is, *without executing them.* An adaptation of σ_7 to ΛM will later be adopted in our $\simeq$ relation (cf. Sec. 7) where (implicit) renaming is left pending as an explicit renaming replacement. In summary, at this stage, our notion of meaningful replacement reduction is taken to be just $\mathtt{R}_{\neg\#}$.

Stage 2: A First Refinement of Stack Replacement. The next stage in the refinement process is to further split the stack replacement rule $\mathtt{R}_{\neg\#}$ based on the number of free occurrences of the name to be replaced, as indicated by $\mathtt{fn}_{\alpha}(c)$ below. The motivation behind this split is that some instances of the replacement rule that replace exactly one name, actually relate terms that are structurally equivalent when translated to PPNs, hence perform no meaningful computation. Thus we consider the following rules:

$$\begin{array}{rcll}
c\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \mapsto_{\mathtt{R}^{\neq 1}_{\neg\#}} & c\{\!\{\alpha\backslash_{\alpha'}s\}\!\} & \text{if } \mathtt{fn}_{\alpha}(c) \neq 1 \text{ and } s \neq \# \\
c\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \mapsto_{\mathtt{R}^{=1}_{\neg\#}} & c\{\!\{\alpha\backslash_{\alpha'}s\}\!\} & \text{if } \mathtt{fn}_{\alpha}(c) = 1 \text{ and } s \neq \#
\end{array}$$

In rule $\mathtt{R}^{\neq 1}_{\neg\#}$ we immediately recognize as involving substantial work due to duplication or erasure of the stack s. Hence, we update our current notion of meaningful replacement computation by replacing our former selection of $\mathtt{R}_{\neg\#}$ with the more restricted $\mathtt{R}^{\neq 1}_{\neg\#}$.

We next have to determine whether rule $\mathtt{R}^{=1}_{\neg\#}$, or refinements thereof, should too be judged as meaningful. For that we must take a closer look at its behavior for specific instances of the command c based on the possible (unique) occurrence of the name α: on a named term (Stage 3) or on an explicit replacement (Stage 4).

Stage 3: Stack Replacement on Named Term. In the right-hand side of rule $\mathtt{R}^{=1}_{\neg\#}$ the command c is traversed by the meta-level replacement $\{\!\{\alpha\backslash_{\alpha'}s\}\!\}$ until the unique free occurrence α is reached. Since free names only occur in commands, the left hand-side of $\mathtt{R}^{=1}_{\neg\#}$ has necessarily one of the following forms:

$$\mathtt{C}\langle[\alpha]\,t\rangle[\![\alpha\backslash_{\alpha'}s]\!] \qquad\qquad \mathtt{C}\langle c'[\![\beta\backslash_{\alpha}s']\!]\rangle[\![\alpha\backslash_{\alpha'}s]\!]$$

for some context $\mathtt{C}$ and where α does not occur free in $\mathtt{C}, t, c', s'$. We next focus on the first case leaving the second to Stage 4. The first case gives rise to the rule `name`:

$$\mathtt{C}\langle[\alpha]\,t\rangle[\![\alpha\backslash_{\alpha'}s]\!] \quad \mapsto_{\mathtt{name}} \quad \mathtt{C}\langle[\alpha']\,t :: s\rangle \quad \text{if } \alpha \notin (t, \mathtt{C}), s \neq \#$$

At this point in our development, we pause and briefly discuss *linear μ-redexes* before getting back to `name`. From the very beginning, there was never any hope for σ-equivalence (Fig. 4) to be a strong bisimulation since, as mentioned by Laurent [23], it does not distinguish between terms with *linear* μ-redexes. A μ-redex in $\lambda\mu$ is *linear* if it has the form $(\mu\alpha.\mathtt{Q}\langle[\alpha]\,u\rangle)\,v$ with $\alpha \notin (u, \mathtt{Q})$ and $\mathtt{Q}$ defined as follows:

$$\mathtt{P} \;::=\; \Box \mid \mathtt{P}\,t \mid \lambda x.\mathtt{P} \mid \mu\alpha.[\beta]\,\mathtt{P} \qquad\qquad \mathtt{Q} \;::=\; \boxdot \mid [\beta]\,\mathtt{P}\langle\mu\gamma.\boxdot\rangle$$

An example is the term $(\mu\alpha.[\alpha]\,x)\,y$, one of the two terms of (3) mentioned in Sec. 1. Such μ-reduction steps reducing linear μ-redexes hold no operational meaning: if o linearly μ-reduces to o', then their graphical interpretation yield PPNs whose multiplicative normal form are structurally equivalent [23, Thm. 41] (revisited in Sec. 8 as Thm. 19).

Returning to our development, we next need to identify what a linear/non-linear split of rule `name` looks like in our setting of ΛM, the intuition arising, again, from PPN. For that we introduce *linear contexts*.

▶ **Definition 6.** *There are four sets of linear contexts, each denoted using the expressions* $\mathtt{XY}$*, with* $\mathtt{X}, \mathtt{Y} \in \{\mathtt{T}, \mathtt{C}\}$*. The letters* $\mathtt{X}$ *and* $\mathtt{Y}$ *in the expression* $\mathtt{XY}$ *denote the sort of the object with which the hole will be filled and the sort of the resulting term, resp.: e.g.* $\mathtt{LTC}$ *denotes a context that takes a command and outputs a term.*

$$\begin{array}{lll}
\textit{\textbf{(Linear TT Contexts)}} & \mathtt{LTT} & ::= \Box \mid \mathtt{LTT}\,t \mid \lambda x.\mathtt{LTT} \mid \mu\alpha.\mathtt{LCT} \mid \mathtt{LTT}[x\backslash t] \\
\textit{\textbf{(Linear TC Contexts)}} & \mathtt{LTC} & ::= \mathtt{LTC}\,t \mid \lambda x.\mathtt{LTC} \mid \mu\alpha.\mathtt{LCC} \mid \mathtt{LTC}[x\backslash t] \\
\textit{\textbf{(Linear CC Contexts)}} & \mathtt{LCC} & ::= \boxdot \mid [\alpha]\,\mathtt{LTC} \mid \mathtt{LCC}[\![\alpha\backslash_{\alpha'}s]\!] \\
\textit{\textbf{(Linear CT Contexts)}} & \mathtt{LCT} & ::= [\alpha]\,\mathtt{LTT} \mid \mathtt{LCT}[\![\alpha\backslash_{\alpha'}s]\!]
\end{array}$$

For example, $[\alpha]\,\boxdot$ is a $\mathtt{LTC}$ context and $[\beta]\,(\Box\,v)[\![\alpha\backslash_{\alpha'}u]\!]$ is a $\mathtt{LTT}$ context.

Given the above definition of linear contexts we can now split `name` into its linear and non-linear versions:

$$\begin{array}{rlll}
\mathtt{C}\langle[\alpha]\,t\rangle[\![\alpha\backslash_{\alpha'}s]\!] & \mapsto_{\mathtt{N}_{\neg\mathtt{lin}}} & \mathtt{C}\langle[\alpha']\,t :: s\rangle & \text{if } \mathtt{C} \text{ not linear}, \alpha \notin (t, \mathtt{C}), s \neq \# \\
\mathtt{LCC}\langle[\alpha]\,t\rangle[\![\alpha\backslash_{\alpha'}s]\!] & \mapsto_{\mathtt{N}} & \mathtt{LCC}\langle[\alpha']\,t :: s\rangle & \text{if } \alpha \notin (t, \mathtt{LCC}),\ s \neq \#
\end{array}$$

The named term $[\alpha]\,t$ in the non-linear rule $\mathtt{N}_{\neg\mathtt{lin}}$ could be duplicated or erased and thus this rule joins $\mathtt{R}^{\neq 1}_{\neg\#}$ as part of meaningful replacement reduction. However, as was the case above for linear μ-redexes, rule $\mathtt{N}$ has no meaningful computational content and hence will be incorporated into $\simeq$ by taking its canonical normal form ($\mathtt{LCC}$ and t below are assumed in canonical normal form). The new equation is called `lin`:

$$\mathtt{LCC}\langle[\alpha]\,t\rangle[\![\alpha\backslash_{\alpha'}s]\!] \simeq_{\mathtt{lin}} \mathtt{LCC}\langle[\alpha']\,\mathfrak{C}(t :: s)\rangle \quad \text{if } \alpha \notin (t, \mathtt{LCC}),\ s \neq \#$$

The notation $\mathfrak{C}(\_)$ denotes the canonical form of an object. A precise definition will be presented in Stage 4.

Summarizing our results of Stage 3, meaningful replacement reduction consists for the moment of $\mathtt{R}^{\neq 1}_{\neg\#}$ and $\mathtt{N}_{\neg\mathtt{lin}}$. In the next and final stage, we analyze the case where the left hand-side of $\mathtt{R}^{=1}_{\neg\#}$ has the form $\mathtt{C}\langle c'[\![\beta\backslash_{\alpha}s']\!]\rangle[\![\alpha\backslash_{\alpha'}s]\!]$.

Stage 4: Stack Replacement on Explicit Replacements. Suppose the left hand-side of $\mathtt{R}^{=1}_{\neg\#}$ has the form $\mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}s'\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}s\rrbracket$. This gives rise to the two instances `swap` and `comp`:

$$\begin{array}{lll}
\mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}\#\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \mapsto_{\mathtt{swap}} \mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}s\rrbracket\llbracket\alpha\backslash_{\alpha'}\#\rrbracket\rangle & \text{if } \alpha\notin(c',\mathtt{C}), s\neq\# \\
\mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}s'\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \mapsto_{\mathtt{comp}} \mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha'}s'\cdot s\rrbracket\rangle & \text{if } \alpha\notin(c',\mathtt{C},s'), s',s\neq\#
\end{array}$$

If we now consider linear/non-linear variants of the above two rules, depending on whether the context $\mathtt{C}$ in their LHSs is a linear context or not (in the sense of Def. 6), we end up with the following four rules, where we use letters r and r' to denote non-empty stacks.

$$\begin{array}{rlll}
\mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}\#\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}r\rrbracket & \mapsto_{\mathtt{W}_{\neg\mathtt{lin}}} & \mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}r\rrbracket\llbracket\alpha\backslash_{\alpha'}\#\rrbracket\rangle & \mathtt{C}\text{ not linear}, \alpha\notin(c',\mathtt{C}) \\
\mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha}r'\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}r\rrbracket & \mapsto_{\mathtt{C}_{\neg\mathtt{lin}}} & \mathtt{C}\langle c'\llbracket\beta\backslash_{\alpha'}r'\cdot r\rrbracket\rangle & \mathtt{C}\text{ not linear}, \alpha\notin(c',\mathtt{C},s') \\
\mathtt{LCC}\langle c'\llbracket\beta\backslash_{\alpha}\#\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}r\rrbracket & \mapsto_{\mathtt{W}} & \mathtt{LCC}\langle c'\llbracket\beta\backslash_{\alpha}r\rrbracket\llbracket\alpha\backslash_{\alpha'}\#\rrbracket\rangle & \alpha\notin(c',\mathtt{LCC}), \mathtt{fc}(\{s,\alpha'\},\mathtt{LCC}) \\
\mathtt{LCC}\langle c'\llbracket\beta\backslash_{\alpha}r'\rrbracket\rangle\llbracket\alpha\backslash_{\alpha'}r\rrbracket & \mapsto_{\mathtt{C}} & \mathtt{LCC}\langle c'\llbracket\beta\backslash_{\alpha'}r'\cdot r\rrbracket\rangle & \alpha\notin(c',\mathtt{LCC},s'), \mathtt{fc}(\{s,\alpha'\},\mathtt{LCC})
\end{array}$$

The rules involving non-linear contexts, namely $\mathtt{W}_{\neg\mathtt{lin}}$ and $\mathtt{C}_{\neg\mathtt{lin}}$ join $\mathtt{R}^{\neq1}_{\neg\#}$ and $\mathtt{N}_{\neg\mathtt{lin}}$ in conforming meaningful replacement computation. Indeed, although there is a unique occurrence of α which is target of the explicit replacement on the LHS of these rules, the stack s could be duplicated or erased. This concludes our deconstruction of rule $\mathtt{R}$.

▶ **Definition 7.** ***Meaningful replacement reduction****, written $\rightarrow_{\mathtt{R}^\bullet}$, is defined by the contextual closure of the reduction rules $\{R^{\neq1}_{\neg\#}, N_{\neg\mathtt{lin}}, \mathtt{W}_{\neg\mathtt{lin}}, \mathtt{C}_{\neg\mathtt{lin}}\}$.*

Rules $\mathtt{M}$, which together with $\mathtt{B}$ compute canonical forms, create new explicit replacements. These explicit replacements may be rearranged using rules $\mathtt{W}$ and $\mathtt{C}$ leading to the following notion of canonical form computation:

▶ **Definition 8.** ***Canonical forms*** *are terms in $\mathtt{B}$, $\mathtt{M}$, $\mathtt{C}$ and $\mathtt{W}$-normal form. The reduction relation $\rightarrow_{\mathtt{BMCW}}$ is easily seen to be confluent and terminating, thus, from now on, the notation $\mathfrak{C}(o)$ stands for the (unique) $\mathtt{BMCW}$-normal form of an object o. It will be shown later that $\mathfrak{C}$-reduction on ΛM-objects corresponds to multiplicative cuts in PPNs (cf. Thm. 14).*

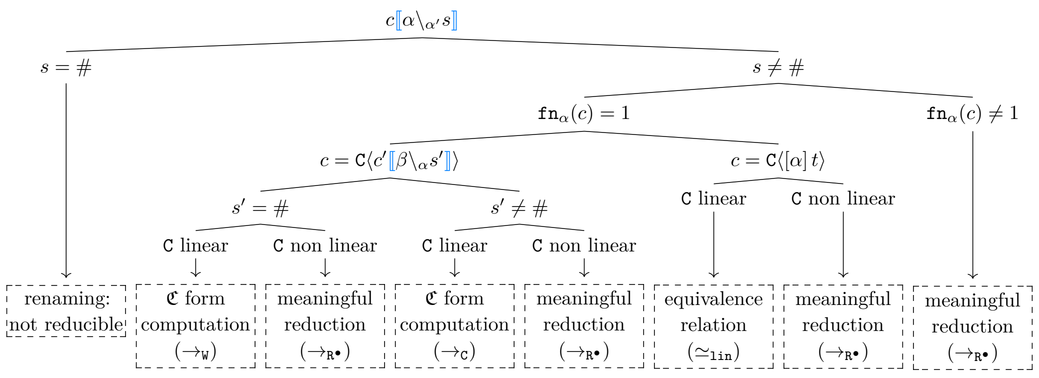
In summary, we shall define a strong bisimulation $\simeq$ (Sec. 7) defined exclusively on canonical forms and where reduction is taken to be meaningful:

▶ **Definition 9.** ***Meaningful reduction*** *is a relation of canonical forms defined as follows:*

$$t \rightsquigarrow t' \text{ iff } t \rightarrow_{\mathtt{SR}^\bullet} u \text{ and } t' = \mathfrak{C}(u)$$

where $\rightarrow_{\mathtt{SR}^\bullet}$ is $\rightarrow_{\mathtt{S}} \cup \rightarrow_{\mathtt{R}^\bullet}$. We may write $\rightsquigarrow_{\mathtt{S}}$ or $\rightsquigarrow_{\mathtt{R}^\bullet}$ to emphasize a meaningful step corresponding to rule $\mathtt{S}$ or $\mathtt{R}^\bullet$ respectively.

The following diagram summarizes this section's findings.


$c\llbracket\alpha\backslash_{\alpha'}s\rrbracket$
$s=\#$
$s\neq\#$
$\mathtt{fn}_\alpha(c)=1$
$\mathtt{fn}_\alpha(c)\neq1$
$c=\mathtt{C}\langle c'\llbracket\beta\backslash_\alpha s'\rrbracket\rangle$
$c=\mathtt{C}\langle[\alpha]\,t\rangle$
$s'=\#$
$s'\neq\#$
C linear
C non linear
C linear
C non linear
C linear
C non linear
renaming: not reducible
$\mathfrak{C}$ form computation ($\rightarrow_{\mathtt{W}}$)
meaningful reduction ($\rightarrow_{\mathtt{R}^\bullet}$)
$\mathfrak{C}$ form computation ($\rightarrow_{\mathtt{C}}$)
meaningful reduction ($\rightarrow_{\mathtt{R}^\bullet}$)
equivalence relation ($\simeq_{\mathtt{lin}}$)
meaningful reduction ($\rightarrow_{\mathtt{R}^\bullet}$)
meaningful reduction ($\rightarrow_{\mathtt{R}^\bullet}$)

$$\frac{}{x : A \vdash x : A \mid \emptyset}\ (\texttt{ax}) \qquad \frac{\Gamma \vdash t : A \to B \mid \Delta \quad \Gamma' \vdash u : A \mid \Delta'}{\Gamma \cup \Gamma' \vdash tu : B \mid \Delta \cup \Delta'}\ (\texttt{app})$$

$$\frac{\Gamma, (x : A)^{\leq 1} \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \to B \mid \Delta}\ (\texttt{abs}) \qquad \frac{\Gamma \vdash c \mid \Delta, (\alpha : A)^{\leq 1}}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\mu) \qquad \frac{\Gamma \vdash t : A \mid \Delta, (\alpha : A)^{\leq 1}}{\Gamma \vdash [\alpha]\, t \mid \Delta, \alpha : A}\ (\texttt{name})$$

$$\frac{\Gamma, (x : B)^{\leq 1} \vdash t : A \mid \Delta \quad \Gamma' \vdash u : B \mid \Delta'}{\Gamma \cup \Gamma' \vdash t[x\backslash u] : A \mid \Delta \cup \Delta'}\ (\texttt{sub})$$

$$\frac{\Gamma \vdash c \mid \Delta, (\alpha : S \to B)^{\leq 1}, (\alpha' : B)^{\leq 1} \quad \Gamma' \vdash s : S \mid \Delta', (\alpha' : B)^{\leq 1}}{\Gamma \cup \Gamma' \vdash c[\![\alpha\backslash_{\alpha'} s]\!] \mid \Delta \cup \Delta', \alpha' : B}\ (\texttt{repl})$$

$$\frac{}{\emptyset \vdash \# : \epsilon \mid \emptyset}\ (\texttt{st}_h) \qquad \frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma' \vdash s : S \mid \Delta'}{\Gamma \cup \Gamma' \vdash t \cdot s : A \cdot S \mid \Delta \cup \Delta'}\ (\texttt{st}_t)$$

Figure 7 Typing Rules for the ΛM-calculus.

5 Types

In this section we introduce simple types for ΛM, which extends the type system in [25] to our syntax. **Types** are generated by the following grammar:

Term Types	A	$::=$	$\iota \mid A \to B$
Stack Types	S	$::=$	$\epsilon \mid A \cdot S$

where ι is a base type. The type constructor $\_\cdot\_$ should be understood as a non-commutative conjunction, which translates to a tensor in linear logic (see [7]). The arrow is right associative. We use the abbreviation $A_1 \cdot A_2 \cdot \ldots \cdot A_n \cdot \epsilon \to B$ for the type $A_1 \to A_2 \ldots \to A_n \to B$ (in particular, $\epsilon \to B$ is equal to B so ϵ is the left neutral element for the functional type). **Variable assignments** (Γ), are functions from variables to types; we write $\emptyset$ for the empty variable assignment. Similarly, **name assignments** (Δ), are functions from names to types. We write $\Gamma \cup \Gamma'$ and $\Delta \cup \Delta'$ for the **compatible union** between assignments meaning that if $x \in \texttt{dom}(\Gamma \cap \Gamma')$ then $\Gamma(x) = \Gamma'(x)$, and similarly for Δ and Δ'. When $\texttt{dom}(\Gamma)$ and $\texttt{dom}(\Gamma')$ are disjoint we may write Γ, Γ'. The same for name assignments.

The **typing rules** are presented in Fig. 7. There are three kinds of **typing judgements**: $\Gamma \vdash t : A \mid \Delta$ for terms, $\Gamma \vdash c \mid \Delta$ for commands and $\Gamma \vdash s : S \mid \Delta$ for stacks. The notation $\Gamma, (x : A)^{\leq 1}$ (resp. $\Delta, (\alpha : A)^{\leq 1}$) is used to denoted either $\Gamma, x : A$ or Γ (resp. either $\Delta, \alpha : A$ or Δ), i.e. the assumption $x : A$ occurs at most once in $\Gamma, (x : A)^{\leq 1}$. Commands have no type, cf. rules (name) and (repl), and stacks are typed with stack types, which are heterogeneous lists, i.e. each component of the list can be typed with a different type. The interesting rule is (repl), which is a logical modus ponens rule, where the fresh variable α' may be already present in the name assignment of the command c or the stack s, thus the notation $\Delta \cup \Delta', \alpha' : B$ means in particular that α' is neither in Δ nor in Δ'.

We use the abbreviation $\Gamma \vdash o : T \mid \Delta$ if $o = t$ and $T = A$, or $o = c$ and there is no type, or $o = s$ and $T = S$. We write $\pi \triangleright \Gamma \vdash o : T \mid \Delta$ if π is a type derivation concluding with $\Gamma \vdash o : T \mid \Delta$. The typing system enjoys the following properties:

▶ **Lemma 10** (Relevance). *Let $o \in \mathcal{O}(\Lambda M)$. If $\pi \triangleright \Gamma \vdash o : T \mid \Delta$, then $\texttt{dom}(\Gamma) = \texttt{fv}(o)$ and $\texttt{dom}(\Delta) = \texttt{fn}(o)$.*

▶ **Lemma 11** (Preservation of Types for $\simeq_\sigma$). *Let $o \in \mathcal{O}(\lambda\mu)$. If $\pi \triangleright \Gamma \vdash o : T \mid \Delta$ and $o \simeq_\sigma o'$, then there exist $\pi' \triangleright \Gamma \vdash o' : T \mid \Delta'$.*

▶ **Lemma 12** (Subject Reduction). *Let $o \in \mathcal{O}(\Lambda M)$ s.t. $\pi_o \triangleright \Gamma \vdash o : T \mid \Delta$. If $o \rightarrow_{\Lambda M} o'$, then there exist $\Gamma' \subseteq \Gamma$ and $\Delta' \subseteq \Delta$ and $\pi_{o'}$ s.t. $\pi_{o'} \triangleright \Gamma' \vdash o' : T \mid \Delta'$.*

Remark that free variables and names of objects decrease in the case of erasing reduction steps, as for example $(\lambda x.y)\, z \rightarrow y$ or $(\mu\alpha.[\gamma]\, x)\, z \rightarrow \mu\alpha.[\gamma]\, x$.

From now on, when $o \simeq_\sigma o'$ (resp. $o \rightarrow_{\Lambda M} o'$), we will refer to π_o and $\pi_{o'}$ as two **related** typing derivations, i.e. $\pi_{o'}$ is obtained from π_o by the proof of Lem. 11 (resp. Lem. 12).

6 Polarized Proof Nets

Laurent [22] introduced Polarized Linear Logic (LLP), a proof system based on polarities on linear logic formulae. It is equipped with a corresponding notion of Polarized Proof-Nets (PPN) which allows for a simpler correctness criterion.

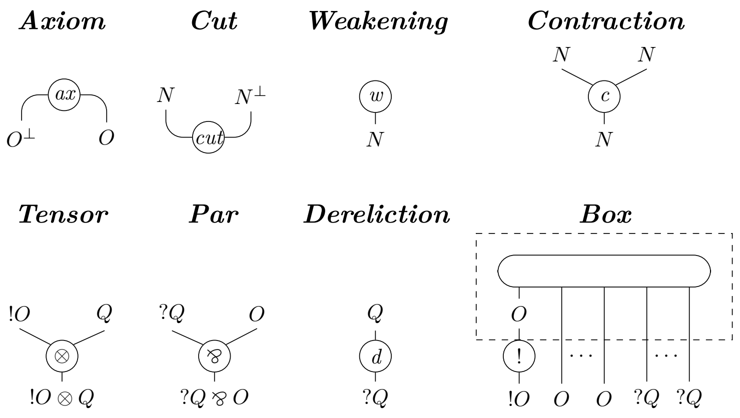
One particularly interesting feature is the translation of classical logic into LLP obtained by interpreting $A \rightarrow B$ as $!A \multimap B$, which is a straightforward extension of that from intuitionistic logic to LLP, thus capturing the translation from λ-calculus to LLP.

As mentioned in Sec. 1, it is possible to translate ΛM-objects to PPNs. More precisely, we translate typing derivations $\pi \triangleright \Gamma \vdash o : T \mid \Delta$. The translations of T and (formulae in) Δ are called *output* formulae whereas, the translations of formulae in Γ are called *input* formulae, given that they correspond to the λ-variables. Following [23], this gives the following formulae categories:

Formulae	F	$::=$	$N \mid P$
Negative formulae	N	$::=$	$O \mid ?Q$
Positive formulae	P	$::=$	$Q \mid !O$
Output formulae	O	$::=$	$\iota \mid ?Q \parr O$
Anti-output formulae	Q	$::=$	$\iota^\perp \mid !O \otimes Q$

Negation is involutive ($\iota^{\perp\perp} = \iota$) with $(?Q \parr O)^\perp = !Q^\perp \otimes O^\perp$ and $(?Q)^\perp = !Q^\perp$.

▶ **Definition 13.** *A **proof-structure** is a finite acyclic oriented graph built over the alphabet of nodes represented below (where the orientation is the top-bottom one):*



Remark that each wire is labelled with a formula. In particular, conclusions in boxes have only one bang formula $!O$, all others being negative formulae.

A **polarized proof-net** (PPN) is a proof-structure satisfying a simple correctness criterion [23]. We refer the reader to op.cit. and simply mention that since our proof-structures are obtained from translating *typed terms*, this criterion is always met. **Structural equivalence** for PPNs, is based on a set of axioms that allow a reordering of weakening and contraction nodes. By lack of space we don't provide all the technical details here, but we refer the interested to [7] for further details on this standard relation. In the sequel, **polarized proof-net equality**, written $\equiv$, is always taken modulo structural equivalence.

The reduction relation for PPNs, denote by $\rightarrow_{\texttt{PPN}}$, is given by a set of *cut elimination rules*, split into *multiplicative* and *exponential* rules. By lack of space we don't include the rules here, but we refer the interested to [7] for further technical details. The major point is that only exponential cuts deal with erasure and duplication of boxes and $\otimes$-trees, these last ones used to interpret stacks in the term language. Exponential cuts are considered as the meaningful rules of PPNs, because of their erasure/duplication power. Multiplicative cuts are confluent and terminating, and we thus use ***multiplicative normal-forms*** as a technical tool to define the translation of typed ΛM-objects to PPNs.

More precisely, the translation from ΛM to PPNs guiding the semantical development of our work is an extension of that introduced in [23] for $\lambda\mu$-terms, based in turn on Girard's translation of classical formulae to linear logic. We do not give any technical detail in this abstract, formal definitions are fully developed in [7]. For the sequel, it is sufficient to keep in mind that formulae are translated to polarized linear logic, and type derivations to PPNs. We use the notation $\pi^{\diamondsuit}$ to denote the translation of the typing derivation π.

Without entering into details, it is worth mentioning that $\pi^{\diamondsuit}$ does not preserve ΛM-reduction, so that we extend it to a new one, written $\_^{\blacklozenge}$, in such a way that $\pi^{\blacklozenge}$ is the multiplicative normal-form of $\pi^{\diamondsuit}$. Then, the following property holds.

▶ **Theorem 14.** *Let o, p be typed ΛM-objects. If $o \rightarrow_{\Lambda M} p$, and π_o, π_p are two related corresponding type derivations for o and p resp., then ${\pi_o}^{\blacklozenge} \twoheadrightarrow_{PPN} {\pi_p}^{\blacklozenge}$.*

Proof. By induction on $\rightarrow_{\Lambda M}$ by adapting Lem. 18 and 19 in [22]. More precisely, the only ΛM-reduction steps involving *exponential* rules on the PPNs side are $\rightarrow_{\texttt{S}}$ and the non-linear instances of $\rightarrow_{\texttt{R}}$ (called $\rightarrow_{\texttt{R}\bullet}$ in Sec. 4), while $\rightarrow_{\texttt{B}}$, $\rightarrow_{\texttt{C}}$ and $\rightarrow_{\texttt{W}}$ are translated to *multiplicative* cuts, and both $\rightarrow_{\texttt{M}}$ and $\rightarrow_{\texttt{N}}$ give the identity. ◀

7 Structural Equivalence for ΛM

We introduce our notion of structural equivalence for ΛM, written $\simeq$, breaking down the presentation into the three key tools on which we have based our development: canonical forms, linear contexts and renaming replacements. Finally, we introduce $\simeq$ itself.

Canonical Forms. As discussed in Sec. 1, the initial intuition in defining a strong bisimulation for ΛM arises from the intuitionistic case: Regnier's equivalence $\simeq_\sigma$ is not a strong bisimulation, but taking the $\texttt{B}$-normal form of the left and right hand sides of these equations, results in a strong bisimulation $\simeq_{\sigma^B}$ on λ-terms with explicit substitutions. In the classical case, we similarly begin from Laurent's $\simeq_\sigma$ relation on $\lambda\mu$-terms and consider the canonical forms, realised by the $\mathfrak{C}$-nf (Def. 8), resulting in the relation $\simeq_{\sigma^B}$ on ΛM-terms (Fig. 8). This equational theory would be the natural candidate for our strong bisimulation, but unfortunately it is not the case as we explain below.

Linear Contexts. The first three equations in Fig. 8 together with equation σ_9^B can be generalized by noting that explicit substitution commutes with *linear* contexts (cf. Def. 6),

$$\begin{array}{rcl}
(\lambda y.t)[x\backslash u] & \simeq_{\sigma_1^B} & \lambda y.t[x\backslash u] \\
(t\, v)[x\backslash u] & \simeq_{\sigma_2^B} & t[x\backslash u]\, v \\
(\mu\alpha.[\beta]\, u)[x\backslash v] & \simeq_{\sigma_3^B} & \mu\alpha.[\beta]\, u[x\backslash v] \\
[\alpha']\,(\mu\alpha''.[\beta']\,(\mu\beta''.c\llbracket\beta\backslash_{\beta''}v\rrbracket)\llbracket\alpha\backslash_{\alpha''}u\rrbracket) & \simeq_{\sigma_4^B} & [\beta']\,(\mu\beta''.[\alpha']\,(\mu\alpha''.c\llbracket\alpha\backslash_{\alpha''}u\rrbracket)\llbracket\beta\backslash_{\beta''}v\rrbracket) \\
[\alpha']\,(\mu\alpha''.([\beta']\,\lambda x.\mu\beta.c)\llbracket\alpha\backslash_{\alpha''}v\rrbracket) & \simeq_{\sigma_5^B} & [\beta']\,\lambda x.\mu\beta.[\alpha']\,(\mu\alpha''.c\llbracket\alpha\backslash_{\alpha''}v\rrbracket) \\
[\alpha']\,\lambda x.\mu\alpha.[\beta']\,\lambda y.\mu\beta.c & \simeq_{\sigma_6^B} & [\beta']\,\lambda y.\mu\beta.[\alpha']\,\lambda x.\mu\alpha.c \\
[\alpha]\,\mu\beta.c & \simeq_{\sigma_7^B} & c\{\beta\backslash\alpha\} \\
\mu\alpha.[\alpha]\, t & \simeq_{\sigma_8^B} & t \\
t[y\backslash v][x\backslash u] & \simeq_{\sigma_9^B} & t[x\backslash u][y\backslash v]
\end{array}$$

Conditions for the equations: $\sigma_1^B : y \notin u$, $\sigma_2^B : x \notin v$, $\sigma_3^B : \alpha \notin v$, $\sigma_4^B : \alpha \notin v, \beta \notin w, \beta'' \neq \alpha', \alpha'' \neq \beta'$, $\sigma_5^B : x \notin v, \beta \notin v, \beta'' \neq \alpha', \alpha'' \neq \beta'$, $\sigma_8^B : \alpha \notin t$, $\sigma_9^B : y \notin u, x \notin v$.

Figure 8 A first reformulation of Laurent's $\simeq_\sigma$ on ΛM-terms.

the latter being the contexts that cannot be erased, nor duplicated, i.e. in proof-net parlance, they do not lay inside a box. The same situation arises between linear contexts and explicit replacements (cf. equations σ_4^B-σ_5^B). Thus, linear contexts can be traversed by any *independent* explicit operator (substitution/replacement). Thanks to linear contexts, equations σ_1^B-σ_2^B-σ_3^B-σ_9^B and also σ_4^B-σ_5^B from Fig. 8 can be subsumed by (and hence replaced with) the commutation between linear contexts and explicit operators as specified by the following equations, which will be part of our equivalence $\simeq$.

$$\begin{array}{rcl}
\mathtt{LTT}\langle v\rangle[x\backslash u] & \simeq_{\mathtt{exs}} & \mathtt{LTT}\langle v[x\backslash u]\rangle \\
\mathtt{LCC}\langle c\rangle\llbracket\alpha\backslash_{\alpha'}s\rrbracket & \simeq_{\mathtt{exr}} & \mathtt{LCC}\langle c\llbracket\alpha\backslash_{\alpha'}s\rrbracket\rangle
\end{array}$$

The first equation is constrained by the condition $x \notin \mathtt{LTT}$ and $\mathtt{fc}(u, \mathtt{LTT})$ while the second one by $\alpha \notin \mathtt{LCC}$ and $\mathtt{fc}(\{s, \alpha'\}, \mathtt{LCC})$, which essentially prevent any capture of free variables/names.

Renaming Replacements. As mentioned in Sec. 4, we adapt $\simeq_{\sigma_7^B}$ by transforming implicit (meta-level) renaming into explicit replacement. The new equation becomes:

$$[\alpha]\,\mu\beta.c \simeq_\rho c\llbracket\beta\backslash_\alpha\#\rrbracket$$

and the situation of the diagram in Fig. 6 is modified as follows:

$$\begin{tikzcd}
([\alpha]\,\mu\beta.c)\llbracket\alpha\backslash_{\alpha'}s\rrbracket \arrow[d, "\mathtt{R}^\bullet"'] \arrow[rr, phantom, "\simeq_\rho"] & & c\llbracket\beta\backslash_\alpha\#\rrbracket\llbracket\alpha\backslash_{\alpha'}s\rrbracket \arrow[d, squiggly, "\mathtt{R}^\bullet"'] \\
{[\alpha']\,(\mu\beta.c\{\!\{\alpha\backslash_{\alpha'}s\}\!\})::s} \arrow[d, "\mathfrak{C}"'] & & \mathfrak{C}(c\llbracket\beta\backslash_\alpha\#\rrbracket\{\!\{\alpha\backslash_{\alpha'}s\}\!\}) \arrow[d, equal, "def"'] \\
{[\alpha']\,\mu\beta'.\mathfrak{C}(c\{\!\{\alpha\backslash_{\alpha'}s\}\!\}\llbracket\beta\backslash_{\beta'}s\rrbracket)} \arrow[rr, phantom, "\simeq_\rho"] & & \mathfrak{C}(c\{\!\{\alpha\backslash_{\alpha'}s\}\!\})\llbracket\beta\backslash_{\beta'}s\rrbracket\llbracket\beta'\backslash_{\alpha'}\#\rrbracket
\end{tikzcd}$$

Note how the behaviour of replacement over renaming replacements (cf. Def. 2) plays a key role in the bottom right corner of the previous diagram.

The Relation $\simeq$ and Admissible Equalities. Given all these considerations, we define:

▶ **Definition 15.** Σ-***equivalence***, *written* $\simeq$, *is a relation over terms in canonical normal form. It is defined as the smallest reflexive, symmetric and transitive relation over terms in canonical normal form, that is closed under the following axioms:*

$$\begin{array}{rcll}
\mathtt{LTT}\langle v\rangle[x\backslash u] & \simeq_{exs} & \mathtt{LTT}\langle v[x\backslash u]\rangle & x\notin \mathtt{LTT}, \mathtt{fc}(u,\mathtt{LTT}) \\
\mathtt{LCC}\langle c\rangle[\![\alpha\backslash_{\alpha'} s]\!] & \simeq_{exr} & \mathtt{LCC}\langle c[\![\alpha\backslash_{\alpha'} s]\!]\rangle & \alpha\notin \mathtt{LCC}, \mathtt{fc}(\{s,\alpha'\},\mathtt{LCC}) \\
([\alpha]\,u)[\![\alpha\backslash_{\alpha'} s]\!] & \simeq_{lin} & [\alpha']\,\mathfrak{C}(u :: s) & \alpha\notin u, s\neq \# \\
[\alpha']\,\lambda x.\mu\alpha.[\beta']\,\lambda y.\mu\beta.u & \simeq_{pp} & [\beta']\,\lambda y.\mu\beta.[\alpha']\,\lambda x.\mu\alpha.u & \alpha\neq\beta', \alpha'\neq\beta \\
[\alpha]\,\mu\beta.c & \simeq_{\rho} & c[\![\beta\backslash_{\alpha}\#]\!] & \\
\mu\alpha.[\alpha]\,t & \simeq_{\theta} & t & \alpha\notin t
\end{array}$$

We conclude this section by showing some interesting *admissible* $\simeq$-equalities. First, we state a permutation result between substitution (resp. replacement) contexts and linear term (resp. command) contexts that will be useful later in the paper:

▶ **Lemma 16.**
1. *Let* $t\in\mathcal{T}(\Lambda M)$. *Then* $\mathfrak{C}(\mathtt{L}\langle\mathtt{LTT}\langle t\rangle\rangle)\simeq\mathfrak{C}(\mathtt{LTT}\langle\mathtt{L}\langle t\rangle\rangle)$, *if* $\mathtt{bv}(\mathtt{L})\notin\mathtt{LTT}$ *and* $\mathtt{fc}(\mathtt{L},\mathtt{LTT})$.
2. *Let* $c\in\mathcal{C}(\Lambda M)$. *Then* $\mathfrak{C}(\mathtt{R}\langle\mathtt{LCC}\langle c\rangle\rangle)\simeq\mathfrak{C}(\mathtt{LCC}\langle\mathtt{R}\langle c\rangle\rangle)$, *if* $\mathtt{bn}(\mathtt{R})\notin\mathtt{LCC}$ *and* $\mathtt{fc}(\mathtt{R},\mathtt{LCC})$.

Some further admissible equations are:

1. $t[x\backslash u][y\backslash v]\simeq t[y\backslash v][x\backslash u]$, where $x\notin v$, and $y\notin u$.
2. $c[\![\alpha'\backslash_{\alpha} s]\!][\![\beta'\backslash_{\beta} s']\!]\simeq c[\![\beta'\backslash_{\beta} s']\!][\![\alpha'\backslash_{\alpha} s]\!]$, where $\alpha\neq\beta'$, $\beta\neq\alpha'$, $\alpha'\notin s'$ and $\beta'\notin s$.
3. $[\alpha']\,\mu\alpha.[\beta']\,\mu\beta.c\simeq[\beta']\,\mu\beta.[\alpha']\,\mu\alpha.c$.

Finally, as already mentioned in Sec. 4, *linear* μ-steps are captured by σ-equivalence [23]. In our setting, they are essentially captured by the axiom $\simeq_{\mathtt{lin}}$ of the $\simeq$-equivalence.

A last remark of this section concerns preservation of types for our equivalence:

▶ **Lemma 17** (Preservation of Types for $\simeq$)**.** *Let* $o\in\mathcal{O}(\Lambda M)$. *If* $\pi\triangleright\Gamma\vdash o:T\mid\Delta$ *and* $o\simeq o'$, *then there exists* $\pi'\triangleright\Gamma\vdash o':T\mid\Delta$.

As before, when $o\simeq o'$ we will refer to π_o and $\pi_{o'}$ as two **related** typing derivations.

8 Two Correspondence Results

We now show how our $\simeq$-equivalence relates to σ, and hence, to PPN equality modulo structural equivalence. In particular, we want to understand whether reshufflings captured by σ-equivalence may have been left out by $\simeq$. One such set of reshufflings are those captured by the equation σ_7^B in Fig. 8. As discussed in Sec. 4 (cf. Fig. 6), this equation breaks strong bisimulation and motivates the introduction of renaming replacements into ΛM, as well as the inclusion of equation $[\alpha]\,\mu\beta.c\simeq_\rho[\![\beta\backslash_\alpha\#]\!]$ into our relation $\simeq$. This gives us:

$$[\alpha]\,\mu\beta.c\simeq_{\sigma_7^B} c\{\beta\backslash\alpha\} \text{ in } \lambda\mu \qquad \text{vs.} \qquad [\alpha]\,\mu\beta.c\simeq_\rho c[\![\beta\backslash_\alpha\#]\!] \text{ in } \Lambda M$$

The question that arises is whether the reshufflings captured by $\simeq_{\sigma_7^B}$ are the only ones that are an obstacle to obtaining a strong bisimulation. We prove in this section that this is indeed the case. This observation is materialized by the following property (cf. Thm. 22):

$$o\simeq_\sigma p \text{ if and only if } \mathfrak{C}(o)\simeq_{\mathtt{er}}\mathfrak{C}(p)$$

where $\simeq_{\mathtt{er}}$ is the **renaming equivalence** generated by our strong bisimulation $\simeq$ plus the following axiom $c\{\!\{\beta\backslash_\alpha\#\}\!\}\simeq_{\mathtt{ren}} c[\![\alpha\backslash_\beta\#]\!]$, which equates the implicit renaming used in equation σ_7^B with the renaming replacement used in equation ρ. This sheds light on the unexpected importance that renaming replacement plays in our strong bisimulation result.

The ($\Rightarrow$) direction of Thm. 22, stated below, is relatively straightforward to prove:

▶ **Lemma 18.** *If $o \simeq_\sigma p$, then $\mathfrak{C}(o) \simeq_{er} \mathfrak{C}(p)$.*

In what follows we focus on the ($\Leftarrow$) direction of Thm. 22. We first discuss the soundness and completeness properties of the equivalence relation $\simeq_{\mathtt{er}}$. This result is based on Laurent's completeness result for σ-equivalence. Indeed, a first translation from typed $\lambda\mu$-objects to polarized proof-nets, written $\_^{\circ}$, is defined in [23], together with a second translation, written $\_^{\bullet}$, which is defined as the multiplicative normal-form of $\_^{\circ}$, where multiplicative normal-forms are obtained by reducing all the so-called multiplicative cuts.

The σ-equivalence relation on $\lambda\mu$-terms has the remarkable property that o and p are σ-equivalent iff their proof-net (second) translation $\_^{\bullet}$ are structurally equivalent (cf. $\equiv$-equivalence introduced in Sec. 6). That is, two σ-equivalent objects have the same *structural* proof-net representation modulo multiplicative cuts.

▶ **Theorem 19** ([23]). *Let o and p be typed $\lambda\mu$-objects such that $o \simeq_\sigma p$ and let π_o, π_p be two related corresponding typing derivations. Then $o \simeq_\sigma p$ iff $\pi_o^{\bullet} \equiv \pi_p^{\bullet}$.*

As mentioned in Sec. 6, we have extended the translation $\_^{\circ}$ to the new constructors of ΛM, the resulting function is written $\_^{\lozenge}$. Moreover, we have also extended the translation $\_^{\lozenge}$ to a new one, written $\_^{\blacklozenge}$, in such a way that $\pi^{\blacklozenge}$ is the multiplicative normal-form of $\pi^{\lozenge}$. Since $\mathcal{O}(\lambda\mu) \subset \mathcal{O}(\Lambda M)$, then for every $\lambda\mu$-object o we have $\pi_o^{\lozenge} \equiv \pi_o^{\circ}$ and $\pi_o{}^{\blacklozenge} \equiv \pi_o^{\bullet}$.

Our relation $\simeq$, together with the equivalence $\simeq_{\mathtt{ren}}$, which are both defined on our calculus ΛM, represent equivalent proof-nets as well:

▶ **Lemma 20** (Soundness). *Let $o, p \in \mathcal{O}(\Lambda M)$ in $\mathfrak{C}$-nf. If $o \simeq_{er} p$, then $\pi_o{}^{\blacklozenge} \equiv \pi_p{}^{\blacklozenge}$, where π_o and π_p are two related corresponding typing derivations.*

Moreover, $\simeq_{\mathtt{er}}$-equivalent terms in $\mathfrak{C}$-nf have an exact correspondence with PPNs.

▶ **Theorem 21** (Correspondence I). *Let $o, p \in \mathcal{O}(\lambda\mu)$. Then $\mathfrak{C}(o) \simeq_{er} \mathfrak{C}(p)$ iff $\pi_{\mathfrak{C}(o)}{}^{\blacklozenge} \equiv \pi_{\mathfrak{C}(p)}{}^{\blacklozenge}$, where $\pi_{\mathfrak{C}(o)}, \pi_{\mathfrak{C}(p)}$ are two related corresponding typing derivations for $\mathfrak{C}(o)$ and $\mathfrak{C}(p)$.*

On the other hand, we have related σ-equivalence in $\lambda\mu$ to Σ-equivalence in ΛM in such a way that $o \simeq_\sigma p$ implies $\mathfrak{C}(o) \simeq_{\mathtt{er}} \mathfrak{C}(p)$ (Lem. 18), where `er` is the equivalence relation generated by converting the renaming replacement into an implicit renaming. The full picture is given by the following result, stating that the converse implication also holds.

▶ **Theorem 22** (Correspondence II). *Let $o, p \in \mathcal{O}(\lambda\mu)$. Then $o \simeq_\sigma p$ iff $\mathfrak{C}(o) \simeq_{er} \mathfrak{C}(p)$.*

The main results of this section can be depicted in the diagram below:

$$\begin{tikzcd}
o \simeq_\sigma p \arrow[r, leftrightarrow, dotted, "Thm.\ 22"] \arrow[d, leftrightarrow, dotted, "Thm.\ 19"'] & \mathfrak{C}(o) \simeq_{\mathtt{er}} \mathfrak{C}(p) \arrow[d, leftrightarrow, dotted, "Thm.\ 21"] \\
\pi_o^{\bullet} \equiv \pi_p^{\bullet} & \pi_{\mathfrak{C}(o)}{}^{\blacklozenge} \equiv \pi_{\mathfrak{C}(p)}{}^{\blacklozenge}
\end{tikzcd}$$

9 The Strong Bisimulation Result

As stated in Thm. 19, $\lambda\mu$-objects that map to the same PPN (modulo structural equivalence) are captured exactly by Laurent's σ-equivalence, which may also been seen as providing a natural relation of *reshuffling*. Unfortunately, as explained in the introduction, σ-equivalence is not a strong bisimulation. We set out to devise a new term calculus in which reshuffling can be formulated as a strong bisimulation *without changing* the PPN semantics. The relation we obtain, $\simeq$, is indeed a strong bisimulation over canonical forms, i.e. $\simeq$-equivalent canonical terms have exactly the same redexes. This is the result we present in this section. It relies crucially on our decomposition of replacement $\rightarrow_{\mathtt{R}}$ (cf. Sec. 4) into *linear* and *non-linear* replacements, the former having no computational content (i.e. structurally equivalent PPNs modulo multiplicative cuts), and thus included in our $\simeq$-equivalence, while the latter corresponding to exponential cut elimination steps, and thus considered as part of our meaningful reduction.

Before stating the bisimulation result, we mention some important technical lemmas:

▶ **Lemma 23.** *Let $o \in \mathcal{O}(\Lambda M)$. If $o \simeq o'$, then $\mathfrak{C}(\mathtt{O}\langle o\rangle) \simeq \mathfrak{C}(\mathtt{O}\langle o'\rangle)$.*

▶ **Lemma 24.** *Let $u, s, o \in \mathcal{O}(\Lambda M)$ be in $\mathfrak{C}$-nf. Assume $p \simeq p'$ and $v \simeq v'$ and $q \simeq q'$. Then,*
- *$\mathfrak{C}(p\{x\backslash u\}) \simeq \mathfrak{C}(p'\{x\backslash u\})$ and $\mathfrak{C}(o\{x\backslash v\}) \simeq \mathfrak{C}(o\{x\backslash v'\})$.*
- *$\mathfrak{C}(p\{\!\{\gamma\backslash_{\gamma'} s\}\!\}) \simeq \mathfrak{C}(p'\{\!\{\gamma\backslash_{\gamma'} s\}\!\})$ and $\mathfrak{C}(o\{\!\{\gamma\backslash_{\gamma'} q\}\!\}) \simeq \mathfrak{C}(o\{\!\{\gamma\backslash_{\gamma'} q'\}\!\})$.*

Proof. Uses Lem. 23. Details in [7]. ◀

We are now able to state the promised result, namely, the fact that $\simeq$ is a strong $\rightsquigarrow$-bisimulation.

▶ **Theorem 25** (Strong Bisimulation). *Let $o \in \mathcal{O}(\Lambda M)$. If $o \simeq p$ and $o \rightsquigarrow o'$, then $\exists p'$ s.t. $p \rightsquigarrow p'$ and $o' \simeq p'$.*

Proof. Uses all the previous lemmas of this section. See [7] for full details. ◀

10 Conclusion

This paper refines the $\lambda\mu$-calculus by splitting its rules into multiplicative and exponential fragments. This new presentation of $\lambda\mu$ allows to reformulate σ-equivalence on $\lambda\mu$-terms as a strong bisimulation relation $\simeq$ on the extended term language ΛM. In addition, $\simeq$ is conservative w.r.t. σ-equivalence, and $\simeq$-equivalent terms share the same PPN representation.

Besides [23], which inspired this paper and has been discussed at length, we briefly mention further related work. In [1], polarized MELL are represented by proof-nets without boxes, by using the polarity information to transform explicit !-boxes into more compact structures. In [19], the $\lambda\mu$-calculus is refined to a calculus $\lambda\mu\mathtt{r}$ with explicit operators, together with a small-step substitution/replacement operational semantics *at a distance*. At first sight $\lambda\mu\mathtt{r}$ seems to be more atomic than ΛM. However, $\lambda\mu\mathtt{r}$ forces the explicit replacements to be evaluated from left to right, as there is no mechanism of composition, and thus only replacements on named locations can be performed. Other refinements of the $\lambda\mu$-calculus were defined in [6, 26, 28]. A further related reference is [16]. A precise correspondence is established between PPN and a typed version of the asynchronous π-calculus. Moreover, they show that Laurent's $\simeq_\sigma$ corresponds exactly to structural equivalence of π-calculus processes (Prop. 1 in op.cit). In [24] Laurent and Regnier show that there is a precise correspondence between CPS translations from classical calculi (such as $\lambda\mu$) into intuitionistic ones on the one hand, and translations between LLP and LL on the other.

Besides confluence, studied in Sec. 2, it would be interesting to analyse other rewriting properties of our term language such as preservation of $\lambda\mu$-strong normalization of the reduction relations $\rightarrow_{\Lambda M}$ and $\rightsquigarrow$, or confluence of $\rightsquigarrow$. Moreover, a reformulation of ΛM in terms of two different syntactical operators, one for renaming replacement, and another one for stack replacements, would probably enlighten the intuitions on PPNs that have been used in this work. We have however chosen a unified syntax for explicit replacements in order to shorten the inductive cases of many of our proofs.

Another further topic would be to explore how our notion of strong bisimulation behaves on different calculi for Classical Logic, such as for example $\lambda\mu\widetilde{\mu}$ [9]. Moreover, following the computational interpretation of deep inference provided by the intuitionistic atomic lambda-calculus [14], it would be interesting to investigate a classical extension and its corresponding notion of strong bisimulation. It is also natural to wonder what would be an ideal syntax for Classical Logic, that is able to capture strong bisimulation by reducing the syntactical axioms to a small and simple set of equations.

We believe the relation $\rightsquigarrow$ is well-suited for devising a residual theory for $\lambda\mu$. That is, treating $\rightsquigarrow$ as an orthogonal system, from a diagrammatic point of view [4], in spite of the critical pairs introduced by ρ and θ. This could, in turn, shed light on call-by-need for $\lambda\mu$ via the standard notion of neededness defined using residuals.

Finally, our notion of $\simeq$-equivalence could facilitate proofs of correctness between abstract machines and $\lambda\mu$ (like [3] for lambda-calculus) and help establish whether abstract machines for $\lambda\mu$ are "reasonable" [3].

References

1 Beniamino Accattoli. Compressing Polarized Boxes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 428–437. IEEE Computer Society, 2013. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6570844`.

2 Beniamino Accattoli. Proof-Nets and the Linear Substitution Calculus. In Bernd Fischer and Tarmo Uustalu, editors, *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, volume 11187 of *Lecture Notes in Computer Science*, pages 37–61. Springer, 2018. `doi:10.1007/978-3-030-02508-3_3`.

3 Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376. ACM, 2014. `doi:10.1145/2628136.2628154`.

4 Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. A nonstandard standardization theorem. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 659–670. ACM, 2014. `doi:10.1145/2535838.2535886`.

5 Beniamino Accattoli and Delia Kesner. The Permutative λ-Calculus. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2012. `doi:10.1007/978-3-642-28717-6_5`.

6 Philippe Audebaud. Explicit Substitutions for the Lambda-Mu-Calculus. Technical report, LIP, ENS Lyon, 1994.

7 Eduardo Bonelli, Delia Kesner, and Andrés Viso. Strong Bisimulation for Control Operators. *CoRR*, abs/1906.09370, 2019. `arXiv:1906.09370`.

8 Roberto Di Cosmo, Delia Kesner, and Emmanuel Polonowski. Proof-Nets and Explicit Substitutions. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures, Third International Conference, FOSSACS 2000, Held as Part of the Joint European*

Conferences on Theory and Practice of Software,ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings, volume 1784 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2000. doi:10.1007/3-540-46432-8_5.

9 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243. ACM, 2000. doi:10.1145/351240.351262.

10 Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space. In *Workshop on Advances in Linear Logic, New York, NY, USA*, page 307–328. Cambridge University Press, 1995.

11 Philippe de Groote. On the Relation between the Lambda-Mu-Calculus and the Syntactic Theory of Sequential Control. In Frank Pfenning, editor, *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94, Kiev, Ukraine, July 16-22, 1994, Proceedings*, volume 822 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 1994. doi:10.1007/3-540-58216-9_27.

12 Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

13 Timothy Griffin. A Formulae-as-Types Notion of Control. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 47–58. ACM Press, 1990. doi:10.1145/96709.96714.

14 Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic Lambda Calculus: A Typed Lambda-Calculus with Explicit Sharing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 311–320. IEEE Computer Society, 2013.

15 Thérèse Hardin. Confluence results for the strong pure categorical logic CCL; λ-calculi as subsystems of CCL. *tcs*, 65, 1989.

16 Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theoretical Computer Science*, 411(22-24):2223–2238, 2010.

17 Delia Kesner. A Theory of Explicit Substitutions with Safe and Full Composition. *Logical Methods in Computer Science*, 5(3), 2009. URL: http://arxiv.org/abs/0905.2539.

18 Delia Kesner and Stéphane Lengrand. Extending the Explicit Substitution Paradigm. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 407–422. Springer, 2005. doi:10.1007/978-3-540-32033-3_30.

19 Delia Kesner and Pierre Vial. Types as Resources for Classical Natural Deduction. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 24:1–24:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.24.

20 Delia Kesner and Pierre Vial. Non-idempotent types for classical calculi in natural deduction style. *Logical Methods in Computer Science*, 2019. To appear.

21 Olivier Laurent. Krivine's abstract machine and the lambda-mu calculus. URL: https://perso.ens-lyon.fr/olivier.laurent/lmkamen.pdf.

22 Olivier Laurent. *A study of polarization in logic.* Theses, Université de la Méditerranée - Aix-Marseille II, March 2002. URL: https://tel.archives-ouvertes.fr/tel-00007884.

23 Olivier Laurent. Polarized proof-nets and lambda-μ-calculus. *Theoretical Computer Science*, 290(1):161–188, 2003.

24 Olivier Laurent and Laurent Regnier. About Translations of Classical Logic into Polarized Linear Logic. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 11–20. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210040.

25 Michel Parigot. Classical Proofs as Programs. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium, KGC'93, Brno, Czech Republic, August 24-27, 1993, Proceedings*, volume 713 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 1993. doi:10.1007/BFb0022575.

26 Emmanuel Polonovski. *Subsitutions explicites, logique et normalisation. (Explicit substitutions, logic and normalization)*. PhD thesis, Paris Diderot University, France, 2004. URL: https://tel.archives-ouvertes.fr/tel-00007962.

27 Laurent Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 2(126):281–292, 1994.

28 Steffen van Bakel and Maria Grazia Vigliotti. A fully-abstract semantics of lambda-mu in the pi-calculus. In Paulo Oliva, editor, *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014.*, volume 164 of *EPTCS*, pages 33–47, 2014. doi:10.4204/EPTCS.164.3.