A Fine-Grained Computational Interpretation of Girard's Intuitionistic Proof-Nets

DELIA KESNER, Université de Paris, CNRS, IRIF, and Institut Universitaire de France, France

This paper introduces a functional term calculus, called pn, that captures the essence of the operational semantics of Intuitionistic Linear Logic Proof-Nets with a faithful degree of granularity, both statically and dynamically. On the static side, we identify an equivalence relation on pn-terms which is sound and complete with respect to the classical notion of structural equivalence for proof-nets. On the dynamic side, we show that every single (exponential) step in the term calculus translates to a different single (exponential) step in the graphical formalism, thus capturing the original Girard's granularity of proof-nets but on the level of terms. We also show some fundamental properties of the calculus such as confluence, strong normalization, preservation of β-strong normalization and the existence of a strong bisimulation that captures pairs of pn-terms having the same graph reduction.

CCS Concepts: • **Theory of computation → Linear logic**.

Additional Key Words and Phrases: lambda-calculus, explicit substitutions, linear logic, proof-nets

ACM Reference Format:
Delia Kesner. 2022. A Fine-Grained Computational Interpretation of Girard's Intuitionistic Proof-Nets. *Proc. ACM Program. Lang.* 6, POPL, Article 8 (January 2022), 28 pages. https://doi.org/10.1145/3498669

1 INTRODUCTION

Pure functional programming can be mathematically understood by means of a universal model of computation, known as the *λ-calculus* [Church 1932] and presenting a clear analogy with *intuitionistic logic* in natural deduction style. This analogy, called the *Curry-Howard isomorphism*, or the *proofs-as-programs* correspondence, emphasizes the fact that proof systems and programming languages are different facets of the same object. It has been extended to many other logical systems like *classical logic* [Griffin 1990] or *type theory* [Martin-Löf 1972], and becomes particularly prolific in the case of linear logic, which is at the center of this paper.

Linear logic (LL), introduced by Jean-Yves Girard [Girard 1987], can be seen as a refinement of classical and intuitionistic logics emphasizing both the role of formulae as *resources* and the linear treatment of information. This is concretely achieved by restricting the use of the structural logical rules of weakening and contraction, which essentially amounts to controlling the crucial operations of *erasure* and *duplication* during proof normalization. Given its emphasis on resources, linear logic has provided a remarkable toolbox for enlightening different disciplines of theoretical computer science, such as for example automata theory, complexity, game semantics and quantum theory, as well as a fruitful range of applications in computational linguistics and programming languages.

The goal of this paper is to reformulate *intuitionistic* LL proof-nets from a programming perspective. Proofs-Nets (PN) are a *graph* formalism proposed by Girard to represent proofs free of *bureaucracy*, *i.e.* the order in which the logical rules are applied in a (sequent) proof derivation is abstracted away. Despite the obvious gain of the graphical interpretation, the essence of its

Author's address: Delia Kesner, Université de Paris, CNRS, IRIF, and Institut Universitaire de France, France, kesner@irif.fr.

2475-1421/2022/1-ART8
https://doi.org/10.1145/3498669

dynamic semantics is not immediately clear from a programming point of view. More generally, the operational semantics of programming languages can be conceptually conceived using graph technology [Asperti et al. 1996; Henderson and Morris 1976; Lamping 1990; Mackie 1998; Muroya and Ghica 2017; Wadsworth 1971], but they are usually specified by a term syntax, which provides a complementary understanding of (bureaucratic-free) graphical frameworks.

Our aim is to define a novel *term language* establishing a *fine-grained* and *faithful* Curry-Howard correspondence with Girard's graphical intuitionistic PN, both from a *static* (objects) and a *dynamic* (reductions) point of view. Providing such a language is not easy, since the degree of granularity on the graphical side cannot be straightforwardly captured by the same degree of granularity on the algebraic side. Let us first discuss some preliminary attempts.

Related Work. Danos and Regnier [Danos 1990; Danos and Regnier 1999; Regnier 1992] were the pioneers in this topic, they showed how cut-elimination in *multiplicative exponential linear logic* (MELL) proof-nets provides a micro-step refinement of β-reduction, the operational semantics of the λ-calculus. Some refinements and extensions were later proposed [Di Cosmo and Kesner 1994; Di Cosmo et al. 2003; Ehrhard and Regnier 2006; Laurent 2003]. The correspondence between the graphical formalism and the algebraic term calculus found in these works can be analyzed under at least two different aspects: a static and a dynamic one.

On the *static* side, λ-terms can be encoded by MELL PN, but the encoding is not injective: two different terms may share the same graph representation. This is not surprising since it is a typical situation arising when translating a (sequential) term syntax into a graphical notation [Muroya and Ghica 2017]. Despite this mismatch, it was shown by [Regnier 1994] that different terms translated to the same MELL PN 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 reductions of the same length. The static correspondence between λ-terms and intuitionistic MELL PN becomes then possible by using a quotient on λ-terms, known as Regnier's σ-equivalence.

On the *dynamic* side, one β-reduction step from a given λ-term can be encoded by several cut elimination steps from its proof-net translation. This encoding puts in evidence the existing gap between the evaluation mechanism in λ-calculus, which is performed by a unique single rule β, and the different kinds of cut-elimination rules in proof-nets: the *multiplicative* ones are operationally simple as they strictly decrease the size of proof-nets by just (linearly) reconfiguring some wires: they can be understood as innocuous. In contrast, the *exponential* ones are non-trivial as they implement erasure and duplication by concatenating several atomic steps: they capture the essential meaning of cut elimination in proof-nets. The granularity of both evaluation procedures, on terms and proof-nets, is clearly not the same, so the challenge is to faithfully capture by a term language what is really happening on the graphical formalism side, and the challenge of a fine-grained simulation clearly lies on the exponential side.

It is at this point that the theory of *explicit substitutions* (ES) [Abadi et al. 1991; de Bruijn 1978; Hardin and Lévy 1989; Kamareddine and Ríos 1995; Nederpelt 1992; Rose 1992] come into the light (a survey can be found in [Kesner 2007]). Some of them are designed by using intuitions coming from category theory [Abadi et al. 1991; Curien 1991; Ghani et al. 1999; Lafont 2019; Ritter 1999]. From a logical point of view, calculi with ES can alternatively be seen as a term notation for cut elimination transformations [Abramsky 1993; Benton et al. 1993; Fernández and Siafakas 2014; Herbelin 1994; Kesner and Lengrand 2005]. They can also bridge the gap between programming languages and graphical formalisms, some examples are [Accattoli 2018; Di Cosmo and Kesner 1997; Milner 2006]. Indeed, ES provide an intermediate formalism that —by decomposing the β-rule into more atomic steps— allows a better understanding of the execution model of functional programs. That is why ES calculi are the basic brick of environment-based abstract machines [Accattoli et al.

2014a; Danos et al. 1996; Danvy and Zerny 2013; Krivine 2007; Landin 1964; Leroy 1990; Sestoft 1997]. In spite of all recent developments in the theory of ES [Accattoli 2018; Accattoli et al. 2014b; Barenbaum and Bonelli 2017], their dynamics is still not fully clear from the perspective of the resource theory provided by Girard's PN.

The State of the Art. Adopting a refined implementation of λ-calculus in view of a perfect correspondence with intuitionistic proof-nets suggests to decompose β-reduction into multiplicative and exponential steps *on the level of terms*. A first attempt gives only two rewriting rules:

$$\begin{array}{lll} (\lambda x.t)u & \mapsto_{\mathsf{B}} & t[x\backslash u] \\ t[x\backslash u] & \mapsto_{\mathsf{subs}} & t\{x\backslash u\} \end{array}$$

where $[x\backslash u]$ is an ES operator, and $\{x\backslash u\}$ is the standard (meta-level) substitution operator. Firing the B-rule creates an ES operator so that B essentially reconfigures symbols, and indeed reads as a series of multiplicative cuts *on the level of proof-nets*. The subs-rule executes the substitution by performing a replacement of all free occurrences of x in t by u, so that it is able to perform erasure/duplication, and reads as a series of exponential cuts *on the level of proof-nets*. However, a single subs-step on the term level potentially translates to a *sequence* of exponential cut-elimination steps on the proof-nets level: we still have a major gap between the term and the graphical side.

Many different term calculi inspired from intuitionistic LL have been defined to investigate and close this gap [Abramsky 1993; Accattoli 2018; Di Cosmo et al. 2003; Ghani et al. 2000; Kesner and Lengrand 2005]. Some of them specify substitution by a big-step operation, while others do implement a more refined small-step semantics. In all cases, none of them achieve a faithful dynamic correspondence with Girard's intuitionistic MELL PN.

One simple way to understand this mismatch is by observing that there are at least two alternative definitions of the meta-level substitution $t\{x\backslash u\}$. The first one, is the usual implementation of substitution by induction on the *structure* of the term t. The second one is given by induction on the *number of free occurrences* of the variable x in the term t: in the base case (0 occurrences of x in t) the ES can be simply erased, in the inductive case ($n > 0$ occurrences of x in t), a *linear* substitution/replacement can be performed, by delaying the $n - 1$ remaining cases. This is exactly the principle used *e.g.* in [de Bruijn 1987; Milner 2006; Severi and Poll 1994], as well as in the *linear substitution calculus (LSC)* by Accattoli and Kesner, a term calculus generalizing Milner's calculus [Milner 2006] with the notion of *reduction at a distance* [Accattoli and Kesner 2010]. However, as we will see, Girard's exponential cut elimination rules do not implement any of these two models of substitution, but a subtle combination of them.

It is also worth mentioning the existence of other static and dynamic correspondences between term algebraic calculi and intuitionistic proof-nets. Some examples:

- Thanks to the use of explicit operators for substitutions, contractions and weakenings, the operational semantics of the λlxr-calculus [Kesner and Lengrand 2005] is very close to that of MELL proof-nets. However, the resulting reduction system (19 reduction rules) becomes cumbersome, and the dynamic simulation of the calculus into proof-nets is not injective.
- The LSC is isomorphic to Accattoli's intuitionistic proof-nets [Accattoli 2011, 2018; Accattoli and Guerrini 2009], where exponential rules for boxes are implemented by *box-crossing*, used to avoid the burden of commutative rules. More precisely, rewriting rules in Accattoli's graphical formalism are based on an *implicit* quantification over boxes used to specify the behavior of cut constructors when crossing a finite number of boxes. The simple and neat *local* definition of reduction in Girard's PN is then converted into a *global* implementation of cut elimination in Accattoli's formalism, which is much heavier to handle.

Thus, the non-trivial question of the definition of a functional programming language faithfully implementing Girard's intuitionistic MELL PN remains still open.

Contributions. We introduce a functional calculus, called pn, which corresponds to Girard's intuitionistic MELL PN on a static (objects) as well as on a dynamic (reduction steps) side. On the static side, we identify an equivalence relation on pn-terms which is sound and complete with respect to the classical notion of structural equivalence on proof-nets (Thm. 6.2 and Thm. 6.5). On the dynamic side, we define an injective translation such that every *single* step in the term calculus translates to a different *single* step in the graphical formalism (Thm. 6.6). This result indicates that pn-reduction has exactly the Girard (exponential) granularity on the dynamic level of terms, a property which does not hold in the framework of all other functional calculi with ES built in the spirit of linear logic.

We also show some healthy untyped and typed properties of the pn-calculus such as full composition (Lem. 3.1), confluence (Thm. 3.4), and strong normalization (Cor. 7.2). A subtle simulation result between the pn-calculus and the structural lambda-calculus modulo permutations [Accattoli and Kesner 2012] provides a non-trivial result of preservation of β-strong normalization (Thm. 8.3). This is a property to be highlighted, since calculi with ES can very easily lose strong normalization [Melliès 1995]. It is also possible to define a strong bisimulation (Thm. 3.5) on the pn-calculus which captures pairs of terms having the same *reduction semantics*.

Our development is based on non-trivial technical points that must be subtly combined. One key technical ingredient to set the definitions of our formalism is the notion of reduction *at a distance* (details in Sec. 2), materialized by the use of two different kinds of *contexts*. Indeed, *head contexts* on the term side represent *linear contexts* in proof-nets —where cut elimination can neither erase nor duplicate—, while *argument contexts* on the term side represent *boxes* in proof-nets, the only kind of structure that can be erased and duplicated during cut elimination. Secondly, the salient nature of our implementation of substitution is given by an operational semantics combining both induction on the structure of terms and induction on the number of free occurrences of variables, in contrast to all other existing implementations of substitution based entirely on one or the other. Another important technical observation allowing reduction on terms to be fine-grained translated to reduction on proof-nets is the use of quotients, both on terms and proof-nets. And the translation used to achieve this result makes abstraction of the multiplicative nature of proof-nets, so that the development is guided by their exponential structure.

Road Map. Sec. 2 introduces the untyped pn-calculus and Sec. 3 shows full composition, confluence and strong bisimulation. A type system for the pn-calculus is presented in Sec. 4. Sec. 5 introduces Linear Logic and MELL proof-nets. In Sec. 6 we present the static and the dynamic translation from pn-terms to MELL PN, together with their corresponding properties. Strong normalization is proved in Sec. 7, and preservation of β-strong normalization is proved in Sec. 8. We conclude in Sec. 9.

2 THE UNTYPED pn-CALCULUS

In this section we introduce the syntax of the pn-calculus. We assume a countable infinite set of symbols $x, y, z, \ldots$. The set of pn-**terms** of the pn-calculus is given by the following grammar:

$$t, u, v ::= x \mid \lambda x.t \mid tu \mid t[x\backslash u]$$

The term x is called a **variable**, tu an **application**, $\lambda x.t$ an **abstraction** and $t[x\backslash u]$ a **closure**. The syntactical object $[x\backslash u]$, which is not a term itself, is called an **explicit substitution** (ES). Terms without closures are λ-terms (also called **pure** terms). Application is left-associative so that $tu_1u_2 \ldots u_n$ $(n \geq 1)$ is an abbreviation of $((tu_1)u_2 \ldots)u_n$.

The sets of **free** and **bound variables of** pn**-terms** are defined as usual, they are denoted respectively by $\mathtt{fv}(t)$ and $\mathtt{bv}(t)$. In particular $\mathtt{fv}(t[x\backslash u]) := (\mathtt{fv}(t) \setminus x) \cup \mathtt{fv}(u)$ and $\mathtt{bv}(t[x\backslash u]) := \mathtt{bv}(t) \cup \{x\} \cup \mathtt{bv}(u)$. We write $|t|$ for the **size** of the term t, defined by $|x| := 1$, $|\lambda x.t| := 1 + |t|$, $|tu| := |t[x\backslash u]| := |t| + |u| + 1$. The **number of free occurrences** of the variable x in the term t is written $|t|_x$, thus *e.g.* $|x[w\backslash x](\lambda y.yx)|_x = 3$.

We work modulo the standard notion of renaming of bound variables given by α**-equivalence** on higher-order terms, thus *e.g.* $\lambda x.x \equiv_\alpha \lambda y.y$ and $x[x\backslash z] \equiv_\alpha y[y\backslash z]$. **Substitution** on pn-terms is defined modulo α-conversion as expected, and written _{__}, thus in particular $t[x\backslash v]\{y\backslash u\} := t\{y\backslash u\}[x\backslash v\{y\backslash u\}]$ is well-defined since we can always assume by α-conversion that the bound variable x is not free in u.

The definition of the operational semantics of the pn-calculus makes use of different notions of (term) **contexts** given by the following grammars:

(**Contexts**)	$\mathtt{C} ::= \mathtt{A} \mid \mathtt{H} \mid \mathtt{L}$
(**Argument Contexts**)	$\mathtt{A} ::= t\Box \mid t[y\backslash\Box]$
(**Head Contexts**)	$\mathtt{H} ::= \Box \mid \lambda x.\mathtt{H} \mid \mathtt{H}t \mid \mathtt{H}[x\backslash t]$
(**List Contexts**)	$\mathtt{L} ::= \Box \mid \mathtt{L}[x\backslash u]$

Even if proof-nets are going to be formally introduced later, we can already give some intuitions about our context definitions. The $\Box$ symbol of an argument context is to be understood as a box in the PN formalism, but $\Box$ in a list context is intended to represent a proof-net linked to a list of cuts, while $\Box$ in a head (term) context is associated to a proof-net which does not lie in any other box of the proof-net (*i.e.* a linear proof-net). Since erasure/duplication in proof-nets does only happen for boxes, then only terms appearing inside (the $\Box$ of) argument contexts can be erased/duplicated in the term formalism, while those appearing inside head contexts are *persistent* (*i.e.* they contribute to the normal form). This is why head contexts are also called *linear* contexts. Notice that list contexts are in particular head contexts.

Free/bound variables are extended to **contexts** as expected. We write $\mathtt{C}\langle t\rangle$ when the symbol $\Box$ in $\mathtt{C}$ is replaced by the term t, and $\mathtt{C}\langle\langle t\rangle\rangle$ when this is done without capturing any free variable of t, *i.e.* there are no abstractions and substitutions in the context $\mathtt{C}$ that bind the free variables in t. For instance, if $\mathtt{H} = \lambda y.\Box$, then $\mathtt{H}\langle y\rangle$ denotes the term $\lambda y.y$, but this last term cannot be written as $\mathtt{H}\langle\langle y\rangle\rangle$ since $\mathtt{H}$ captures the free variable y. We can however write $\lambda y.z$ as $\mathtt{H}\langle\langle z\rangle\rangle$ for $y \neq z$. A term t is said to be **free for a context** $\mathtt{C}$, written $\mathtt{fc}(t, \mathtt{C})$ if $\mathtt{bv}(\mathtt{C}) \cap \mathtt{fv}(t) = \emptyset$. Thus *e.g.* x is free for $\lambda y.\Box$ but y is not free for $\lambda y.\Box$. Given an argument context $\mathtt{A}$, we write $\mathtt{A}_{[x\backslash u]}$ to denote a **substituted argument context** defined as follows: $t[x\backslash u]\Box$ if $\mathtt{A} = t\Box$, and $t[x\backslash u][y\backslash\Box]$ if $\mathtt{A} = t[y\backslash\Box]$.

Operational Semantics. We now define the operational semantics of the pn-calculus, which can be seen as an implementation of β-reduction, given by the rewriting rule $(\lambda x.t)u \mapsto t\{x\backslash u\}$. In order to incrementally introduce the rewriting rules of our framework, let us analyze different situations.

Firing β. The rule that *fires* β-reduction takes in principle the natural form $(\lambda x.t)u \mapsto t[x\backslash u]$. However, when working in a syntax equipped with closures, this rule could be blocked by an explicit substitution lying between a function $\lambda x.t$ and its argument u, *e.g.* in $(\lambda x.t)[y\backslash v]u$. This kind of stuck redex does not happen in graphical representations such as proof-nets (*e.g.* [Girard 1996a]), but it is typical in the sequential structure of term syntaxes. There are at least two ways to handle this issue. The first one is based on *structural/permutation* rules, as in [Gundersen et al. 2013]. Indeed, in the previous example, the substitution is first pushed outside the application node by means of a permutation rule, as $(\lambda x.t)[y\backslash v]u \mapsto ((\lambda x.t)u)[y\backslash v]$, so that firing β-reduction is finally possible. The second, less elementary, possibility is given by an operational semantics at

a distance, inherited from the *structural lambda-calculus* [Accattoli and Kesner 2010], where the β-rule can be fired by a rule like $\mathtt{L}\langle\lambda x.t\rangle u \mapsto \mathtt{L}\langle t[x\backslash u]\rangle$, where $\mathtt{L}$ is an arbitrary (capture-free) list context. The distant paradigm is therefore used to gather meaningful and permutation rules in only one reduction step called distant Beta, and written dB.

Erasing. Another rule is necessary to deal with useless substitutions, *i.e.* when $x \notin \mathtt{fv}(t)$ in a term of the form $t[x\backslash u]$. The erasure step is then performed by the rule $t[x\backslash u] \mapsto t$, called garbage collector and written gc. This is the only rule of the calculus having erasure power.

Substituting the Head. The rewriting rule that fires a substitution on a single variable takes in principle the form $x[x\backslash u] \mapsto u$. However, this operation could be blocked by another ES, or more generally by a head context, as *e.g.* in $(\lambda z.x)[y\backslash v][x\backslash u]$. Again, permutation like rules could be used to push the head context $(\lambda z.\Box)[y\backslash v]$ outside, as in $(\lambda z.x)[y\backslash v][x\backslash u] \mapsto (\lambda z.x[x\backslash u])[y\backslash v]$ so that the elementary reduction that substitutes x becomes now possible. Our calculus implements these two operations by using again the distant paradigm, this is done in one single step of the form $\mathtt{H}\langle x\rangle[x\backslash u] \mapsto \mathtt{H}\langle u\rangle$, where $\mathtt{H}$ is an arbitrary (capture-free) head context. We call this rule hs because it is implementing head (linear) substitution.

Jumping into arguments. As explained before, the $\Box$ symbol of an argument context will be associated to a box in MELL proof-nets. Since boxes explicitly handle erasure and duplication in linear logic, they play a key role in our reduction rules. Thus, if the variable x to be substituted is essentially inside an argument context, *i.e.* the term to be substituted is of the form $\mathtt{A}\langle t\rangle$, where $x \in \mathtt{fv}(t)$ and $x \notin \mathtt{fv}(\mathtt{A})$, then the explicit substitution can in principle be pushed inside the argument by a rule like $\mathtt{A}\langle t\rangle[x\backslash u] \mapsto \mathtt{A}\langle t[x\backslash u]\rangle$. However, as in the case of the hs rule, some other explicit substitutions —or more generally a head context— could lie between the term to be substituted $\mathtt{A}\langle t\rangle$ and the ES $[x\backslash u]$, as *e.g.* in $(yx)[z\backslash v][x\backslash u]$. Instead of using permutation like rules to push the substitution $[z\backslash v]$ outside the whole term, we use again the paradigm at a distance by introducing a rule to jump inside the argument t of $\mathtt{A}$. The rule, called arg, is of the form $\mathtt{H}\langle\mathtt{A}\langle t\rangle\rangle[x\backslash u] \mapsto \mathtt{H}\langle\mathtt{A}\langle t[x\backslash u]\rangle\rangle$.

Duplicating. Another situation arises when an explicit substitution affects different subterms of a term, typically one inside an argument context and another one outside this context. This duplicating situation is captured by means of contraction nodes in proof-nets, but it appears to be only implicit in the term syntax of our calculus. Since we have already defined a rule to jump inside an argument, we would now need a rule to jump inside the complementary (non-argument) part of an argument context, something like $\mathtt{A}\langle t\rangle[x\backslash u] \mapsto \mathtt{A}_{[x\backslash u]}\langle t\rangle[x\backslash u]$. Notice that the explicit substitution $[x\backslash u]$ needs to be duplicated in this case, in contrast to all the other rules presented above. Following the same idea developed before for the other rules of the calculus, we also consider this rule modulo *distance*, by allowing a head context to lie between the argument and the explicit substitution. The rule, called dup is written $\mathtt{H}\langle\mathtt{A}\langle t\rangle\rangle[x\backslash u] \mapsto_{\mathtt{dup}} \mathtt{H}\langle\mathtt{A}_{[x\backslash u]}\langle t\rangle[x\backslash u]\rangle$.

In summary, the operational semantics of the pn-calculus is based on a reduction relation generated by the following set of pn-**rewriting rules**:

$$
\begin{array}{llll}
\mathtt{L}\langle\lambda x.t\rangle u & \mapsto_{\mathtt{dB}} & \mathtt{L}\langle t[x\backslash u]\rangle & \\
t[x\backslash u] & \mapsto_{\mathtt{gc}} & t & x \notin \mathtt{fv}(t) \\
\mathtt{H}\langle\langle x\rangle\rangle[x\backslash u] & \mapsto_{\mathtt{hs}} & \mathtt{H}\langle\langle u\rangle\rangle & x \notin \mathtt{fv}(\mathtt{H}) \\
\mathtt{H}\langle\mathtt{A}\langle t\rangle\rangle[x\backslash u] & \mapsto_{\mathtt{arg}} & \mathtt{H}\langle\mathtt{A}\langle t[x\backslash u]\rangle\rangle & x \notin \mathtt{fv}(\mathtt{H}), x \in \mathtt{fv}(t), x \notin \mathtt{fv}(\mathtt{A}) \\
\mathtt{H}\langle\mathtt{A}\langle t\rangle\rangle[x\backslash u] & \mapsto_{\mathtt{dup}} & \mathtt{H}\langle\mathtt{A}_{[x\backslash u]}\langle t\rangle[x\backslash u]\rangle & x \notin \mathtt{fv}(\mathtt{H}), x \in \mathtt{fv}(t), x \in \mathtt{fv}(\mathtt{A})
\end{array}
$$

By analogy with linear logic proof-nets, we call the rule dB **multiplicative**, while all the remaining rules $\{\mathtt{gc}, \mathtt{hs}, \mathtt{arg}, \mathtt{dup}\}$ are called **exponential**. This nomenclature will become clear in Sec. 6.2.

The reader will notice that given $t_0 = (xt)[x\backslash u]$, then $x \notin \mathtt{fv}(t)$ implies t_0 can be written as $\mathtt{H}\langle x\rangle[x\backslash u]$ where $\mathtt{H} = \Box t$ and $x \notin \mathtt{fv}(\mathtt{H})$ (so that rule hs applies); while $x \in \mathtt{fv}(t)$ implies t_0 can be written as $\mathtt{A}\langle t\rangle[x\backslash u]$, where $\mathtt{A} = x\Box$ (so that rule dup applies). More generally, it can be shown that any term $t[x\backslash u]$ corresponds to *exactly one* of the left-hand sides among the rules {gc, hs, arg, dup}.

Our implementation of substitution is given by an operational semantics of substitution combining *both* induction on the structure of terms, and induction on the number of free occurrences of variables, in contrast to all other existing implementations of substitution based entirely on one or the other. Indeed, take a term of the form $t[x\backslash u]$. If there is no free occurrence of x in t, then rule gc applies, while one free *head* (linear) occurrence of x in t makes use of rule hs, and *non-head* free occurrence(s) of x in t are treated by *structurally* propagating the substitution $[x\backslash u]$ along the term t according to the rules arg and dup.

Given $\mathtt{r} \in \{\mathtt{dB}, \mathtt{hs}, \mathtt{gc}, \mathtt{arg}, \mathtt{dup}\}$, we write $\to_{\mathtt{r}}$ for the reduction relation generated by the rewriting rule $\mapsto_{\mathtt{r}}$ and closed under *all* contexts. The pn-**reduction relation**, written $\to_{\mathtt{pn}}$ or simply $\to$, is the union of all such relations. The generic notation $\to_{r_1,\dots,r_n}$ means $\to_{r_1} \cup \ldots \cup \to_{r_n}$.

Here is an example of $\to_{\mathtt{pn}}$-reduction sequence, where we highlight in *blue* the contexts used to close the rewriting rules, and in *green* (resp. *orange*) the list/head (resp. argument) contexts used in the rewriting rules.

$$
\begin{array}{llllll}
(\,(\lambda z.\lambda y.\lambda x.yxx)w\;u)v & \to_{\mathtt{dB}} & (\,(\lambda y.\lambda x.yxx)[z\backslash w]\;u)v & \to_{\mathtt{gc}} & & \\
(\,(\lambda y.\lambda x.yxx)u\;)v & \to_{\mathtt{dB}} & (\lambda x.yxx)\;[y\backslash u]\;v & \to_{\mathtt{dB}} & & \\
((\,y\;x)x)[x\backslash v]\;[y\backslash u] & \to_{\mathtt{hs}} & ((ux)\;x\,)\;[x\backslash v] & \to_{\mathtt{dup}} & & \\
((ux)[x\backslash v]\;x\,)\;[x\backslash v] & \to_{\mathtt{arg}} & ((ux)[x\backslash v]\;x[x\backslash v]\,) & \to_{\mathtt{hs}} & & \\
(\;(u\;x\,)\;[x\backslash v]\;v) & \to_{\mathtt{arg}} & ((u\;x[x\backslash v]\,)v) & \to_{\mathtt{hs}} & ((uv)v) &
\end{array}
$$

As expected, free variables are preserved by reduction.

Lemma 2.1. *Let t be a term.*

- *If $t \to_{\mathtt{dB,hs,arg,dup}} u$, then $\mathtt{fv}(t) = \mathtt{fv}(u)$.*
- *If $t \to_{\mathtt{gc}} u$, then $\mathtt{fv}(t) \supseteq \mathtt{fv}(u)$.*

3 PROPERTIES

We now discuss and show some of the main untyped properties of the pn-calculus: full-composition, confluence and strong bisimulation.

3.1 Full Composition

Full composition (FC) is a very natural property which guarantees that any term can be substituted as such, without any need to compute its value beforehand. ES calculi without any sort of composition operation for substitutions (*e.g.* the λx-calculus [Bloo and Rose 1995]) do not verify the FC property, but the pn-calculus, as well as other ES calculi using reduction at a distance, do verify it. In what follows, we present the full proof of FC which gives a good understanding of the precise role played by each rewrite rule of the pn-calculus in view of the implementation of substitution.

Lemma 3.1 (Full Composition). *For all pn-terms t, u, we have $t[x\backslash u] \to^*_{\mathtt{pn}} t\{x\backslash u\}$.*

Proof. We prove a more general property, namely, for all pn-terms t, u and all head context $\mathtt{H}$ such that $\mathtt{fc}(u, \mathtt{H})$ and $x \notin \mathtt{fv}(\mathtt{H})$ we have $\mathtt{H}\langle t\rangle[x\backslash u] \to^*_{\mathtt{pn}} \mathtt{H}\langle t\rangle\{x\backslash u\}$. We proceed by induction on t.

- $t = y$. If $y \neq x$, then $\mathtt{H}\langle y\rangle[x\backslash u] \to_{\mathtt{gc}} \mathtt{H}\langle y\rangle = \mathtt{H}\langle y\rangle\{x\backslash u\}$. If $y = x$, then $\mathtt{H}\langle x\rangle[x\backslash u] \to_{\mathtt{hs}} \mathtt{H}\langle u\rangle = \mathtt{H}\langle x\rangle\{x\backslash u\}$.

- $t = \lambda y.v$. Then let $\mathsf{H}' = \mathsf{H}\langle\lambda y.\Box\rangle$. The *i.h.* gives $\mathsf{H}\langle\lambda y.v\rangle[x\backslash u] = \mathsf{H}'\langle v\rangle[x\backslash u] \to_{\mathsf{pn}}^{*} \mathsf{H}'\langle v\rangle\{x\backslash u\} = \mathsf{H}\langle\lambda y.v\rangle\{x\backslash u\}$.
- $t = v_1 v_2$. There are four cases:
  - $x \notin \mathtt{fv}(v_1)$ and $x \notin \mathtt{fv}(v_2)$. Then $\mathsf{H}\langle v_1 v_2\rangle[x\backslash u] \to_{\mathsf{gc}} \mathsf{H}\langle v_1 v_2\rangle = \mathsf{H}\langle v_1 v_2\rangle\{x\backslash u\}$.
  - $x \in \mathtt{fv}(v_1)$ and $x \notin \mathtt{fv}(v_2)$. Let $\mathsf{H}' = \mathsf{H}\langle\Box v_2\rangle$. The *i.h.* gives $\mathsf{H}\langle v_1 v_2\rangle[x\backslash u] = \mathsf{H}'\langle v_1\rangle[x\backslash u] \to_{\mathsf{pn}}^{*} \mathsf{H}'\langle v_1\rangle\{x\backslash u\} = \mathsf{H}\langle v_1 v_2\rangle\{x\backslash u\}$.
  - $x \notin \mathtt{fv}(v_1)$ and $x \in \mathtt{fv}(v_2)$. Using the *i.h.* with an empty head context we obtain $v_2[x\backslash u] \to_{\mathsf{pn}}^{*} v_2\{x\backslash u\}$. Thus $\mathsf{H}\langle v_1 v_2\rangle[x\backslash u] \to_{\mathsf{arg}} \mathsf{H}\langle v_1 v_2[x\backslash u]\rangle \to_{\mathsf{pn}}^{*} (i.h.)\ \mathsf{H}\langle v_1 v_2\{x\backslash u\}\rangle = \mathsf{H}\langle v_1 v_2\rangle\{x\backslash u\}$.
  - $x \in \mathtt{fv}(v_1)$ and $x \in \mathtt{fv}(v_2)$. Using the *i.h.* with empty head contexts we obtain $v_1[x\backslash u] \to_{\mathsf{pn}}^{*} v_1\{x\backslash u\}$ and $v_2[x\backslash u] \to_{\mathsf{pn}}^{*} v_2\{x\backslash u\}$. Then $\mathsf{H}\langle v_1 v_2\rangle[x\backslash u] \to_{\mathsf{dup}} \mathsf{H}\langle(v_1[x\backslash u]v_2)[x\backslash u]\rangle \to_{\mathsf{pn}}^{*} (i.h.)\ \mathsf{H}\langle(v_1\{x\backslash u\}v_2)[x\backslash u]\rangle \to_{\mathsf{arg}}^{*} \mathsf{H}\langle(v_1\{x\backslash u\}v_2[x\backslash u])\rangle \to_{\mathsf{pn}}^{*} (i.h.)\ \mathsf{H}\langle(v_1\{x\backslash u\}v_2\{x\backslash u\})\rangle = \mathsf{H}\langle v_1 v_2\rangle\{x\backslash u\}$.
- $t = v_1[y\backslash v_2]$. There are also four cases, and the proof is similar to the previous case.

□

Here is an example illustrating the previous lemma. Let $t_0 := (xwx)[x\backslash y[y\backslash z]]$. Then,

$$\begin{array}{llllll} t_0 & \to_{\mathsf{dup}} & ((xw)[x\backslash y[y\backslash z]]x)[x\backslash y[y\backslash z]] & \to_{\mathsf{arg}} & (xw)[x\backslash y[y\backslash z]]x[x\backslash y[y\backslash z]] \\ & \to_{\mathsf{hs}} & (xw)[x\backslash y[y\backslash z]]y[y\backslash z] & \to_{\mathsf{hs}} & (y[y\backslash z]w)y[y\backslash z]c = (xwx)\{x/y[y\backslash z]\} \end{array}$$

3.2 Confluence

Confluence is a property stating that although terms can, in principle, be rewritten in more than one way, they ultimately give the same result. It is a fundamental property guaranteeing *determinism*, in the sense that every terminating program has one and only one result.

In order to show confluence of the pn-calculus we use the *interpretation method* [Hardin and Lévy 1989], a technique which allows highlighting the relation between the pn-calculus and the λ-calculus[1]. The first part of the proof is based on the fact that β-reduction can be implemented by pn-reduction.

Lemma 3.2 (Simulation). *Let t be a λ-term. If $t \to_{\beta} t'$, then $t \to_{\mathsf{pn}}^{+} t'$.*

Proof. By induction on β-reduction on λ-terms using the Full Composition Lemma 3.1. □

The second part of the confluence proof is devoted to show that pn-reduction can be projected into β-reduction, a property which is achieved by the following transformation of pn-terms:

$$\begin{array}{lcl} \mathtt{proj}(x) & := & x \\ \mathtt{proj}(\lambda y.t) & := & \lambda y.\mathtt{proj}(t) \\ \mathtt{proj}(tu) & := & \mathtt{proj}(t)\mathtt{proj}(u) \\ \mathtt{proj}(t[x\backslash u]) & := & \mathtt{proj}(t)\{x\backslash \mathtt{proj}(u)\} \end{array}$$

Thus for example, $\mathtt{proj}((xy)[x'\backslash z'][x\backslash z][z\backslash w[w\backslash w']]) = w'y$.

Lemma 3.3 (Projection). *Let t, u be* pn*-terms.*

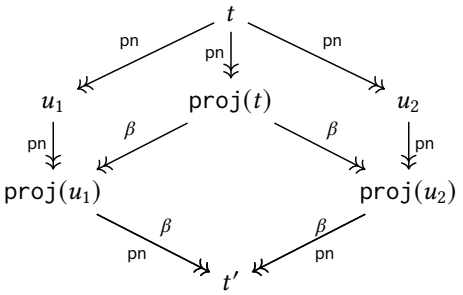
(1) $\mathtt{proj}(t)\{x\backslash\mathtt{proj}(u)\} = \mathtt{proj}(t\{x\backslash u\})$.
(2) $t \to_{\mathsf{pn}}^{*} \mathtt{proj}(t)$.
(3) *If* $t \to_{\mathsf{hs,gc,arg,dup}} u$, *then* $\mathtt{proj}(t) = \mathtt{proj}(u)$.
(4) *If* $t \to_{\mathsf{dB}} u$, *then* $\mathtt{proj}(t) \to_{\beta}^{*} \mathtt{proj}(u)$.
(5) *If* $t \to_{\mathsf{pn}} u$, *then* $\mathtt{proj}(t) \to_{\beta}^{*} \mathtt{proj}(u)$.

[1] Nevertheless, other proof techniques seem to be possible, notably strong commutation [Terese 2003].

Proof. Point (1) is shown by induction on t. Point (2) is by induction on t using the Full Composition Lemma 3.1. Points (3) and (4) are shown by induction on the relations $\to_{\mathsf{dB}}$ and $\to_{\mathsf{pn}}$ respectively, and use the first point. The last point is a consequence of the two previous points. □

Theorem 3.4. *The* pn*-calculus is confluent.*

Proof. Let $t \to_{\mathsf{pn}}^* u_1$ and $t \to_{\mathsf{pn}}^* u_2$. By the Projection Lemma 3.3(5) we have $\mathtt{proj}(t) \to_\beta^* \mathtt{proj}(u_i)$ $(i = 1, 2)$. Since β-reduction is confluent then there exists v such that $\mathtt{proj}(u_i) \to_\beta^* v$. Then $\mathtt{proj}(u_i) \to_{\mathsf{pn}}^+ t'$ holds by the Simulation Lemma 3.2. We conclude since $u_i \to_{\mathsf{pn}}^* \mathtt{proj}(u_i)$ $(i = 1, 2)$ holds by the Projection Lemma 3.3(2). Graphically,



□

3.3 Strong Bisimulation

This section is about terms having the same reduction semantics. This notion can be captured by the mathematical notion of *strong bisimulation*: it is a symmetric relation $\simeq$ such that $t \simeq u$ and $t \to t'$ imply the existence of u' such that $u \to u'$ and $t' \simeq u'$. Graphically,

$$\begin{array}{ccc} t & \simeq & u \\ \downarrow & & \downarrow \\ t' & \simeq & u' \end{array}$$

The formulation of strong bisimulations for term calculi is inevitably related to their sequential aspect. Consider for example the terms $(\lambda x.(\lambda y.t)u)v$ and $(\lambda x.\lambda y.t)vu$. They seem to have the same redexes, only permuted, however, this is not entirely correct. The former has two redexes (one indicated by underlining and another by overlining) $\underline{(\lambda x.\overline{(\lambda y.t)u})v}$, and the latter has only one (underlined) $\underline{(\lambda x.(\lambda y.t))vu}$. The overlined redex in the first term is not visible in the second one; it will only reappear, as a newly created redex, once the underlined redex is computed.

Despite the sequential nature of λ-calculus, these terms behave in essentially the same way [Regnier 1994]. More precisely, Regnier 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 σ-equivalent terms $(\lambda x.(\lambda y.t)u)v$ and $(\lambda x.(\lambda y.t))vu$ is still unsatisfying since there clearly seems to be an underlying strong bisimulation, which is not showing itself due to the sequential notation of the formalism itself. Thanks to the graphical intuition provided by LL PN, the term syntax of the λ-calculus enriched with explicit substitutions unveils a strong bisimulation in the LSC for the intuitionistic case [Accattoli et al. 2014b]. In this paper, we resort to this same intuition to uncover a strong bisimulation for the pn-calculus.

We adopt an equivalence relation used in a classical setting [Kesner et al. 2020] , which simplifies here to the following scheme of **equations** specifying the permeability of head contexts w.r.t. ES:

$$\mathsf{H}\langle t\rangle[x\backslash u] \equiv \mathsf{H}\langle t[x\backslash u]\rangle \text{ if } x \notin \mathtt{fv}(\mathsf{H}) \text{ and } \mathtt{fc}(u,\mathsf{H})$$

where H is a head context, t and u are pn-terms and x is variable. Notice that the conditions of the equational scheme avoids capture of free variables of the term u by the context H. We write $\simeq_\sigma$ for the **congruence** (reflexive, symmetric, transitive, closed by contexts) relation generated by α-conversion and the equations $\equiv$. Thus in particular, the following equalities —known as call-by-name σ-equivalence for ES [Accattoli and Kesner 2012]— are valid in our framework:

$$\begin{array}{lcll} t[y\backslash v][x\backslash u] & \simeq_\sigma & t[x\backslash u][y\backslash v] & \text{if } x \notin \mathtt{fv}(v) \text{ and } y \notin \mathtt{fv}(u) \\ (\lambda y.t)[x\backslash u] & \simeq_\sigma & \lambda y.t[x\backslash u] & \text{if } y \notin \mathtt{fv}(u) \\ (tv)[x\backslash u] & \simeq_\sigma & t[x\backslash u]v & \text{if } x \notin \mathtt{fv}(v) \end{array}$$

The congruence $\simeq_\sigma$ relates terms that translate to (structural) equivalent proof-nets (*c.f.* Thm. 6.2). It seems then unavoidable the use of quotients on terms in order to match the same graphical object: terms are sequential while proof-nets were precisely defined to remove the bureaucracy of sequential notations. See also the full discussion before Thm. 6.2.

Theorem 3.5 (Strong Bisimulation). *The relation $\simeq_\sigma$ is a strong bisimulation w.r.t. $\to_{\mathtt{pn}}$: if $t \simeq_\sigma u$ and $t \to t'$, then there is u' such that $u \to u'$ and $t' \simeq_\sigma u'$.*

It is worth noticing that this key result could not be achieved by a weaker reduction system lacking the notion of distance for head contexts. Indeed, suppose we replace the pn-calculus with the following system *without distance*:

$$\begin{array}{llll} (\lambda x.t)u & \mapsto_{\mathsf{dB}'} & t[x\backslash u] & \\ t[x\backslash u] & \mapsto_{\mathtt{gc}} & t & x \notin \mathtt{fv}(t) \\ x[x\backslash u] & \mapsto_{\mathtt{hs}'} & u & \\ \mathsf{A}\langle t\rangle[x\backslash u] & \mapsto_{\mathtt{arg}'} & \mathsf{A}\langle t[x\backslash u]\rangle & x \in \mathtt{fv}(t), x \notin \mathtt{fv}(\mathsf{A}) \\ \mathsf{A}\langle t\rangle[x\backslash u] & \mapsto_{\mathtt{dup}'} & \mathsf{A}_{[x\backslash u]}\langle t\rangle[x\backslash u] & x \in \mathtt{fv}(t), x \in \mathtt{fv}(\mathsf{A}) \end{array}$$

Here are some examples of diagrams that cannot be closed:

- $uy \;{}_{\mathtt{hs}'}\!\leftarrow x[x\backslash u]y \simeq_\sigma (xy)[x\backslash u]$.
- $(yx[x\backslash u])z \;{}_{\mathtt{arg}'}\!\leftarrow (yx)[x\backslash u]z \simeq_\sigma ((yx)z)[x\backslash u]$.
- $(x[x\backslash u]x)[x\backslash u]z \;{}_{\mathtt{dup}'}\!\leftarrow (xx)[x\backslash u]z \simeq_\sigma ((xx)z)[x\backslash u]$.

This is one of the key points where head contexts and reduction at a distance come into play.

4 THE TYPED pn-CALCULUS

This section presents a typed version of the pn-calculus, which will be related in Sec. 6 to proof-nets. **Types** are given by the grammar $A, B ::= \iota \mid A \Rightarrow B$, where ι belongs to a set of *atomic types*. An **environment** is a finite function from variables to types. The **support** of an environment Γ is given by $\mathsf{supp}(\Gamma) := \{x \mid \Gamma(x) \text{ is defined}\}$. We write $\Gamma \subseteq \Delta$ if $\mathsf{supp}(\Gamma) \subseteq \mathsf{supp}(\Delta)$ and $x : A \in \Gamma$ implies $\mathsf{x} : A \in \Delta$. Two environments Γ and Δ are said to be **compatible** iff $x = y$ implies $A = B$ for all $x : A \in \Gamma$ and all $y : B \in \Delta$. The **union of compatible contexts** is written $\Gamma \cup \Delta$. Thus for example $(x : A, y : B) \cup (x : A, z : C) = (x : A, y : B, z : C)$. The predicate $\Gamma\#\Delta$ means $\mathsf{supp}(\Gamma) \cap \mathsf{supp}(\Delta) = \emptyset$. When $\Gamma\#\Delta$ we write Γ, Δ instead of $\Gamma \cup \Delta$. **Typing judgments** have the form $\Gamma \vdash t : A$ where t is a term, A is a type and Γ is an environment. The typing system for the pn-calculus is presented in Fig. 1, where the typing rules read naturally from top to bottom. Notice: (1) the absence of weakening, which implies the use of two disjoint rules for the cases of abstraction and closure, and (2) the use of multiplicative rules for the cases of closure and

$$\frac{}{x : A \vdash x : A}\,(\mathsf{var}) \qquad \frac{\Gamma \vdash t : B \Rightarrow A \quad \Delta \vdash u : B}{\Gamma \cup \Delta \vdash t\ u : A}\,(\mathsf{app})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\,(\mathsf{abs}_{\in}) \qquad \frac{\Gamma \vdash u : B \quad \Delta, x : B \vdash t : A}{\Gamma \cup \Delta \vdash t[x/u] : A}\,(\mathsf{es}_{\in})$$

$$\frac{\Gamma \vdash t : B \quad x \notin \mathsf{supp}(\Gamma)}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\,(\mathsf{abs}_{\notin}) \qquad \frac{\Gamma \vdash u : B \quad \Delta \vdash t : A \quad x \notin \mathsf{supp}(\Delta)}{\Gamma \cup \Delta \vdash t[x/u] : A}\,(\mathsf{cut}_{\notin})$$

Fig. 1. Typing Rules for the pn-Calculus

$$\frac{}{x : A \vdash x : A}\,(\mathsf{var}) \qquad \frac{\Gamma, \Pi \vdash t : B \Rightarrow A \quad \Delta, \Pi \vdash u : B \quad \Gamma \# \Delta}{\Gamma, \Delta, \Pi \vdash t\ u : A}\,(\mathsf{app})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\,(\mathsf{abs}_{\in}) \qquad \frac{\Gamma, \Pi \vdash u : B \quad \Delta, \Pi, x : B \vdash t : A \quad \Gamma \# \Delta}{\Gamma, \Delta, \Pi \vdash t[x/u] : A}\,(\mathsf{es}_{\in})$$

$$\frac{\Gamma \vdash t : B \quad x \notin \mathsf{supp}(\Gamma)}{\Gamma \vdash \lambda x.t : A \Rightarrow B}\,(\mathsf{abs}_{\notin}) \qquad \frac{\Gamma, \Pi \vdash u : B \quad \Delta, \Pi \vdash t : A \quad x \notin \mathsf{supp}(\Delta, \Pi) \quad \Gamma \# \Delta}{\Gamma, \Delta, \Pi \vdash t[x/u] : A}\,(\mathsf{cut}_{\notin})$$

Fig. 2. Equivalent Formulation of the Typing Rules for the pn-Calculus

application. As a consequence, the system turns out to be relevant (*c.f.* Lem. 4.1). An alternative but equivalent formulation of the typing system is presented in Fig. 2, where in each binary rule, the overlap between the left and right environments (denoted by Π) is made explicit, so that the additive (resp. multiplicative) treatment of common (resp. non-common) variables is highlighted. This presentation is more appropriate for the inductive proofs, which are carried out from bottom to top. **(Type) derivations**, written $\Gamma \vdash_{\mathsf{pn}} t : A$, can be obtained by application of a finite number of typing rules. A term is said to be **typable** iff there is a type derivation such that $\Gamma \vdash_{\mathsf{pn}} t : A$.

Here is an example of typing derivation, where $A = B \Rightarrow ((D \Rightarrow B) \Rightarrow C)$:

$$\dfrac{\dfrac{}{y' : A \vdash y' : A}\,(\mathsf{var}) \qquad \dfrac{\dfrac{\dfrac{}{y : A \vdash y : A}\,(\mathsf{var}) \qquad \dfrac{}{x : B \vdash x : B}\,(\mathsf{var})}{y : A, x : B \vdash yx : (D \Rightarrow B) \Rightarrow C}\,(\mathsf{app}) \qquad \dfrac{\dfrac{}{x : B \vdash x : B}\,(\mathsf{var})}{x : B \vdash \lambda z.x : D \Rightarrow B}\,(\mathsf{abs}_{\notin})}{y : A, x : B \vdash yx(\lambda z.x) : C}\,(\mathsf{app})}{y' : A, x : B \vdash (yx(\lambda z.x))\,[y \backslash y'] : C}\,(\mathsf{es}_{\in})$$

Lemma 4.1 (Relevance). *If* $\Gamma \vdash_{\mathsf{pn}} t : A$, *then* $\mathsf{supp}(\Gamma) = \mathsf{fv}(t)$.

Proof. By induction on $\Gamma \vdash_{\mathsf{pn}} t : A$. □

Theorem 4.2 (Subject Reduction). *If* $\Gamma \vdash_{\mathsf{pn}} t : A$ *and* $t \to_{\mathsf{pn}} t'$, *then* $\Gamma' \vdash_{\mathsf{pn}} t' : A$ *for some* $\Gamma' \subseteq \Gamma$.

Proof. We prove the property by induction on $t \to_{\mathsf{pn}} t'$ by showing in particular the following:

- $t \simeq_{\sigma} t'$ implies $\Gamma \vdash t' : A$.

- $t \to_{\mathsf{dB,hs,arg,dup}} t'$ implies $\Gamma \vdash t' : A$.
- $t \to_{\mathsf{gc}} t'$ implies $\Gamma' \vdash t' : A$ for some $\Gamma' \subseteq \Gamma$.

The proof is routine and needs also induction on head contexts, but to illustrate the idea, we detail here a simple case for $\to_{\mathsf{dup}}$, where reduction takes place inside the context $\Box$, and the head context is also $\Box$. We consider the argument context to be an application, so that we have $t = (v_1 v_2)[x\backslash u] \to_{\mathsf{dup}} (v_1[x\backslash u]v_2)[x\backslash u] = t'$ with $x \in \mathsf{fv}(v_i)$ $(i = 1, 2)$. We start by considering a type derivation for $(v_1 v_2)[x\backslash u]$:

$$\dfrac{\dfrac{\Gamma_{v_1}, x : B \vdash v_1 : C \Rightarrow A \qquad \Gamma_v, x : B \vdash v_2 : C}{\Gamma_{v_1} \cup \Gamma_{v_2}, x : B \vdash v_1 v_2 : A}\,(\mathsf{app}) \qquad \Gamma_u \vdash u : B}{(\Gamma_{v_1} \cup \Gamma_{v_2}) \cup \Gamma_u \vdash (v_1 v_2)[x\backslash u] : A}\,(\mathsf{es}_\in)$$

where $\Gamma = \Gamma_{v_1} \cup \Gamma_{v_2} \cup \Gamma_u$. Then we construct the following derivation for t':

$$\dfrac{\dfrac{\dfrac{\Gamma_{v_1}, x : B \vdash v_1 : C \Rightarrow A \qquad \Gamma_u \vdash u : B}{\Gamma_{v_1} \cup \Gamma_u \vdash v_1[x\backslash u] : C \Rightarrow A}\,(\mathsf{es}_\in) \qquad \Gamma_{v_2}, x : B \vdash v_2 : C}{(\Gamma_{v_1} \cup \Gamma_u) \cup \Gamma_{v_2}, x : B \vdash v_1[x\backslash u]v_2 : A}\,(\mathsf{es}_\in) \qquad \Gamma_u \vdash u : B}{((\Gamma_{v_1} \cup \Gamma_u) \cup \Gamma_{v_2}) \cup \Gamma_u \vdash (v_1[x\backslash u]v_2)[x\backslash u] : A}\,(\mathsf{es}_\in)$$

We can conclude since $(\Gamma_{v_1} \cup \Gamma_{v_2}) \cup \Gamma_u = ((\Gamma_{v_1} \cup \Gamma_u) \cup \Gamma_{v_2}) \cup \Gamma_u$. $\Box$

5 MULTIPLICATIVE EXPONENTIAL LINEAR LOGIC AND PROOF-NETS

The core connectives of linear logic are divided into two groups: *additives* and *multiplicatives*, each of them introduced by the conclusion of a logical rule having a finite number of premises. For the multiplicative connectives, the context/hypothesis of the conclusion is split up between the premises, whereas for the additive cases the context of the conclusion is entirely transported into both premises. The conjunction of classical logic splits into the additive $\&$ (with) and the multiplicative $\otimes$ (tensor) connectives. Similarly, the classical disjunction is divided into an additive $\oplus$ (plus) and a multiplicative $\parr$ (par) connectives. To recover the full expressive power of intuitionistic and classical logics, it is also necessary to allow *weakening* and *contraction* structural rules. Introducing them implicitly would collapse the system into classical logic itself, so they are introduced in a controlled way, by means of two dual *modalities*, which are called resp. the *of-course* ! and the *why-not* ? *exponentials*. Erasure (resp. duplication) of formulae can then only be materialized by cut-elimination rules acting on exponential formulae introduced by weakening (resp. contraction). All these ingredients lead to the formulation of full propositional linear logic.

Multiplicative exponential linear logic (MELL) is the fragment of full linear logic obtained by omitting the additive connectives. In this paper we use MELL as a synonymous of MELL *without units*, which is sufficient in particular to encode pure functional programming, corresponding to *minimal* intuitionistic logic by means of the Curry-Howard isomorphism. We recall here some of its basic notions, and we refer the interested reader to [Girard 1987] for a general and more detailed introduction to full linear logic.

Let us consider a set of *atomic symbols* of the form ι and $\underline{\iota}$. The set of formulae of MELL is defined by the following grammar:

$$A ::= \iota \mid \underline{\iota} \mid A \parr B \mid A \otimes B \mid ?A \mid !A$$

The *linear negation* of a MELL formula A, denoted $A^\perp$ is defined by the following De Morgan equations:

$$\begin{array}{llllllllll} \iota^\perp & := & \underline{\iota} & \qquad (A \parr B)^\perp & := & A^\perp \otimes B^\perp & \qquad (?A)^\perp & := & !A^\perp \\ \underline{\iota}^\perp & := & \iota & \qquad (A \otimes B)^\perp & := & A^\perp \parr B^\perp & \qquad (!A)^\perp & := & ?A^\perp \end{array}$$

Notice that the linear negation is involutive: $A^{\perp\perp} = A$.

We write $\vdash \Gamma$ for a **(unilateral) sequent**, where Γ is a list of formulae. If Γ is the list $A_1, \dots A_m$, we denote by $?\Gamma$ the list $?A_1, \dots, ?A_m$ and by $\Gamma^\perp$ the list ${A_1}^\perp, \dots, {A_m}^\perp$. The logical system MELL is given by the rules in Fig. 3:

$$\frac{}{\vdash A^\perp, A}\,(\mathtt{Id}) \qquad \frac{\vdash A, \Gamma \qquad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta}\,(\mathtt{Cut}) \qquad \frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}\,(\mathtt{X}) \qquad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}\,(\parr) \qquad \frac{\vdash A, \Gamma \qquad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta}\,(\otimes)$$

$$\frac{\vdash \Gamma}{\vdash ?A, \Gamma}\,(\mathtt{Weak}) \qquad \frac{\vdash ?A, ?A, \Gamma}{\vdash ?A, \Gamma}\,(\mathtt{Cont}) \qquad \frac{\vdash A, \Gamma}{\vdash ?A, \Gamma}\,(?) \qquad \frac{\vdash A, ?\Gamma}{\vdash !A, ?\Gamma}\,(!)$$

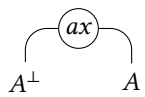
Fig. 3. MELL Rules

Notice that the two binary rules (Cut) and ($\otimes$) are multiplicative. Weakening and contraction are restricted to ?-formulae, and rule (?) induces the loss of linearity for the underlying formula. Rule (!) is only possible when the context contains only ?-formulae, and allows the main formula to become potentially erased or duplicated.

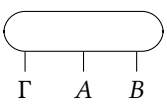
MELL Proof-Nets. Proof-nets were introduced by Jean-Yves Girard [Girard 1996b] as an abstract representation of linear logic proofs, the goal being to create a unique representation for syntactical different (tree) proofs that are morally the same. It is in this sense that proof-nets are said to be a graphical formalism eliminating syntactical *bureaucracy*.

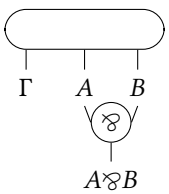
In general, a MELL proof-net can be seen as a finite acyclic oriented graph (see the discussion after Thm. 6.2) verifying some correctness criteria which ensures that the graph actually encodes a MELL sequent derivation. For the purpose of this section, we prefer the alternative presentation which defines proof-nets as graphs given by an inductive definition based on the sequent rules in Fig. 3. For each of these rules, there is a different proof-net construction, except for (X) which vanishes in the graph notation: the wires of a graph are naturally considered to be unordered. In what follows, we define the set of **MELL proof-nets**, that we denote by PN. Each MELL proof-net has an associated multiset of conclusions, each of which is a MELL formula.

- For every MELL formula A, there is a PN with conclusions $A, A^\perp$ having the form:

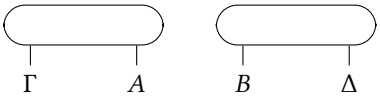


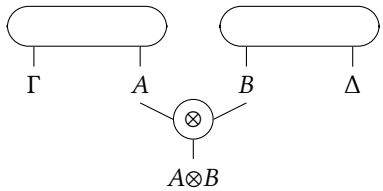
- Given a PN with conclusions Γ, A, B on the left we can construct the following PN with conclusions $\Gamma, A \parr B$ on the right



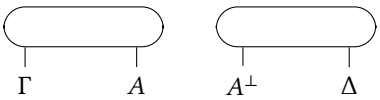


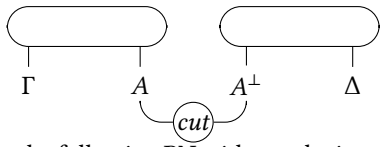
- Given a PN with conclusions Γ, A and a PN with conclusions Δ, B, both on the left, we can construct the following PN with conclusions $\Gamma, A \otimes B, \Delta$ on the right





- Given a PN with conclusions Γ, A and a PN with conclusions $\Delta, A^{\perp}$, both on the left, we can construct the following PN with conclusions Γ, Δ on the right





- Given a PN with conclusions Γ on the left, we can construct the following PN with conclusions $\Gamma, ?A$ on the right

- Given a PN with conclusions $\Gamma, ?A, ?A$ on the left, we can construct the following PN with conclusions $\Gamma, ?A$ on the right

- Given a PN with conclusions Γ, A on the left, we can construct the following PN with conclusions $\Gamma, ?A$ on the right

- Given a PN with conclusions $?\Gamma, A$ on the left, we can construct the following PN with conclusions $?\Gamma, !A$ on the right

Notice that squares with rounded corners denote arbitrary proof-nets, while the box constructor (which is a particular proof-net) is denoted with a dotted line square, so that boxes are used to *encapsulate* other proofs, which can now be potentially erased or duplicated. Axioms may be restricted to atomic symbols by using a notion of eta-like expansion on proof-nets, but we prefer to keep in this paper the traditional general presentation.

The **cut elimination rules** for MELL appear in Fig. 4. Rules $\{\mathtt{C}(a), \mathtt{C}(⅋, ⊗)\}$ are known as **multiplicative**, while $\{\mathtt{C}(w, b), \mathtt{C}(d, b), \mathtt{C}(c, b), \mathtt{C}(b, b)\}$ are called **exponential** (these cuts only relates exponential formulae). As mentioned in the introduction, the *multiplicative* rules are essentially used to (linearly) reconfigure wires, and thus they are harmless. In contrast, the *exponential* ones implement erasure and duplication, so they capture the crucial operational semantics of MELL.

The reduction relation generated by the multiplicative rules $\{\mathtt{C}(a), \mathtt{C}(⅋, ⊗)\}$ is trivially confluent and terminating, so we can consider (unique) $\mathtt{C}(a)/\mathtt{C}(⅋, ⊗)$**-normal forms** of proof-nets in the sequel (see in particular Sec. 7). They are also called **multiplicative normal forms**.

Although the cut elimination rules in Fig. 4 are sufficient to describe the normalization process of proofs from a logical point of view, the original MELL formalism by Girard also needs some complementary equations in order to capture/simulate the notion of evaluation in functional programs (see *e.g.* [Laurent 2002]). In fact, it is necessary to identify those proof-nets which, while having the same *box structure* (same nesting and number of boxes), allow certain liberal movements of their weakening and contraction nodes. We thus consider the four equations in Fig. 5. The first equations $\mathtt{E}(c, c)$ specifies associativity of contraction nodes, the second one $\mathtt{E}(b, c)$ axiomatizes permeability of boxes w.r.t. contraction, the third one $\mathtt{E}(c, w)$ specifies neutrality of weakening w.r.t. the contraction operation, the fourth one $\mathtt{E}(b, w)$ pushes final weakening nodes to the top level.

We call $\mathcal{R}$ (resp. $\mathcal{E}$) the set of the following rewriting rules (resp. axioms)

$$\begin{array}{rcl} \mathcal{R} & := & \{\mathtt{C}(a), \mathtt{C}(⅋, ⊗), \mathtt{C}(w, b), \mathtt{C}(d, b), \mathtt{C}(c, b), \mathtt{C}(b, b)\} \\ \mathcal{E} & := & \{\mathtt{E}(c, c), \mathtt{E}(b, c), \mathtt{E}(c, w), \mathtt{E}(b, w)\} \end{array}$$

The reduction relation $\rightarrow_{\mathcal{R}}$ is defined as the closure by *all* the *PN* contexts of the rewriting rules in $\mathcal{R}$. The contextual relation $\widehat{\mathcal{E}}$ is defined as the closure of the equations $\mathcal{E}$ in the following way: the identities $\mathtt{E}(c, c)$, $\mathtt{E}(b, c)$ and $\mathtt{E}(c, w)$ are closed by any context, while $\mathtt{E}(b, w)$ is only closed by contexts not binding the weakening wire (*i.e.* not connecting the weakening wire to any other construct). Finally, the **structural equivalence** $\simeq_{\mathcal{E}}$ on MELL proof-nets is taken as the reflexive, symmetric and transitive closure of $\widehat{\mathcal{E}}$. In the sequel, MELL proof-nets are always considered modulo structural equivalence $\simeq_{\mathcal{E}}$.

We shall write $\rightarrow_{\mathcal{R}/\mathcal{E}}$ for the reduction relation on MELL PN generated by $\rightarrow_{\mathcal{R}}$ modulo the equivalence relation $\simeq_{\mathcal{E}}$, *i.e.*

$$p \rightarrow_{\mathcal{R}/\mathcal{E}} p' \text{ iff } \exists p_1, p_2 \text{ such that } p \simeq_{\mathcal{E}} p_1 \rightarrow_{\mathcal{R}} p_2 \simeq_{\mathcal{E}} p'$$

The reduction relation $\rightarrow_{\mathcal{R}/\mathcal{E}}$ on MELL PN is known to be strongly normalizing [Di Cosmo and Piperno 1995], *i.e.* every $\rightarrow_{\mathcal{R}/\mathcal{E}}$-reduction sequence starting at a MELL PN is finite.

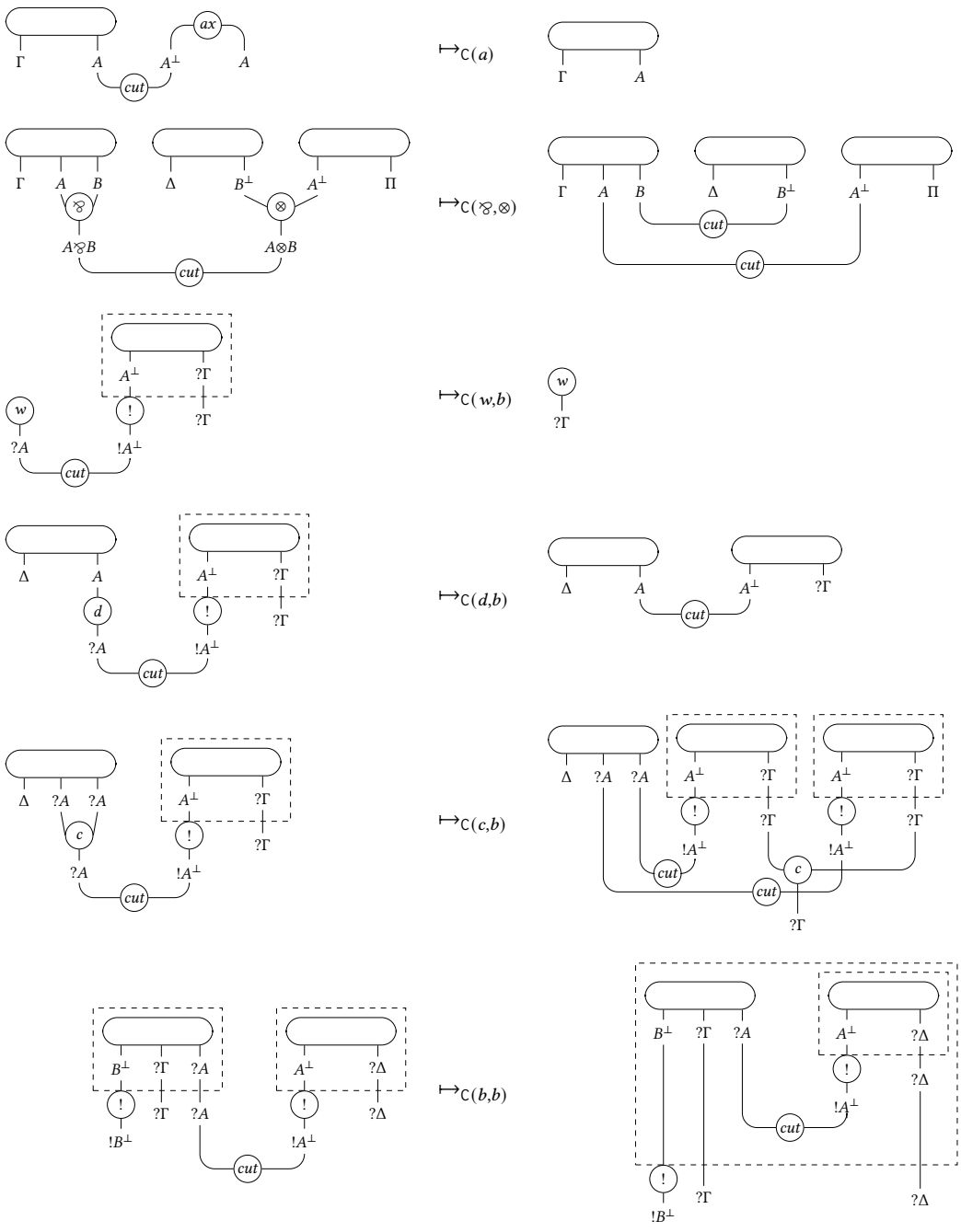


Fig. 4. Cut Elimination for MELL Proof-Nets

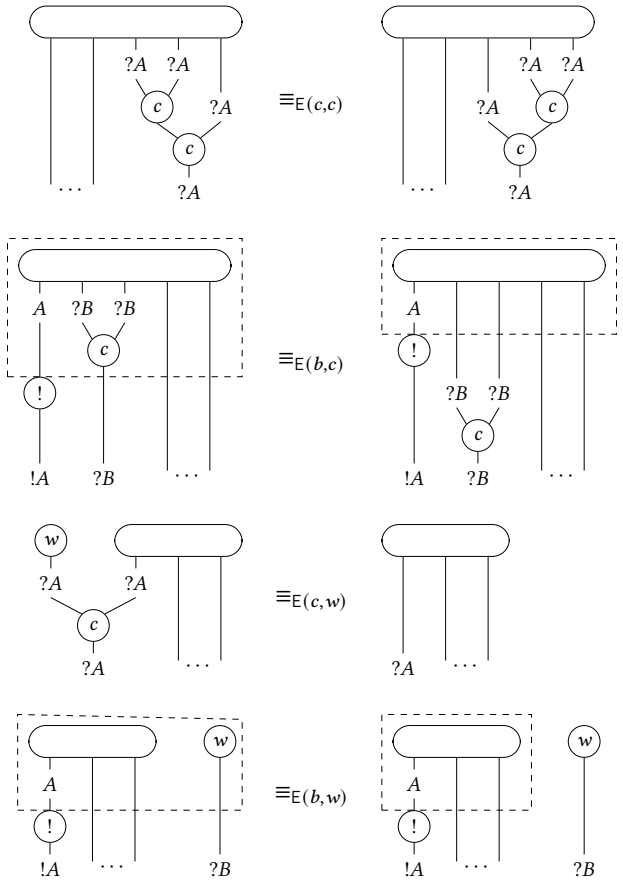


Fig. 5. Equations $\mathcal{E}$ for MELL Proof-Nets

6 TRANSLATING THE TYPED pn-CALCULUS INTO MELL PROOF-NETS

This section presents a translation of the typed pn-calculus into MELL proof-nets. The *static* translation is discussed in Sec. 6.1, where simple types and typing derivations (*c.f.* Sec. 4) are encoded into MELL formulae and MELL proof-nets (*c.f.* Sec. 5) respectively. Soundness and completeness results of this encoding are proved. The *dynamic* translation is presented in Sec. 6.2: it is shown that pn-reduction can be simulated by the reduction relation $\rightarrow_{\mathcal{R}/\mathcal{E}}$ on proof-nets (*c.f.* Sec. 5). In particular, one single exponential $\rightarrow_{\mathsf{pn}}$-step on pn-term is captured by exactly one different single exponential $\rightarrow_{\mathcal{R}/\mathcal{E}}$-step on proof-nets (see the discussion after Thm. 6.6).

6.1 The Static Translation

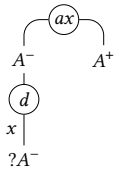
This section presents the translation from pn-terms to proof-nets, together with appropriate results of soundness and completeness of the translation. For that, let's start by the usual translation of intuitionistic types [Girard 1987] into MELL formulae given by :

$$\begin{array}{lcl} \iota^{+} & := & \iota \\ (A \to B)^{+} & := & ?(A^{-}) \parr B^{+} \\ A^{-} & := & (A^{+})^{\perp} \end{array}$$

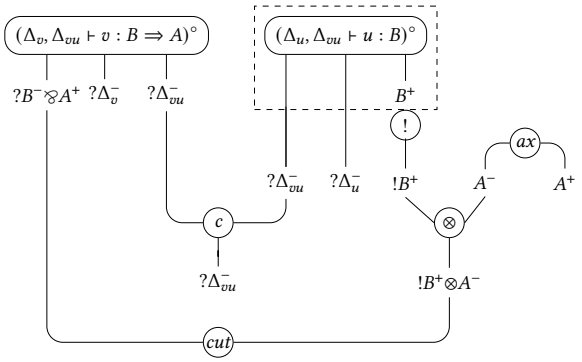
The translation of (typed) pn-terms into MELL proof-nets is defined by induction on typing derivations. In particular, given a derivation $x_1 : B_1, \ldots, x_n : B_n \vdash_{\mathsf{pn}} t : A$, the resulting proof-net contains n conclusions of the form $?B_1^-, \ldots, ?B_n^-$, and one conclusion for A^+. We also add a **label** x_i to each conclusion $?B_i^-$ of the resulting proof-net. The conclusion A^+ has no name/label. A variable which is erased during some inductive step of the translation, is written between parentheses. Proof-net equality and reduction take variable labels into account. In particular, we would like the translation of the type derivations $x : A \vdash_{\mathsf{pn}} x : A$ and $y : A \vdash_{\mathsf{pn}} y : A$ to result in different proof-nets, although the resultant proof-nets actually have the same graphical structure and they only differ in their labels. As mentioned above, labels between parenthesis are assumed to have been erased and hence are not taken into account in proof-net equality/reduction.

Given an environment $\Gamma = x_1 : B_1, \ldots, x_n : B_n$ we write $?\Gamma^-$ for the multiset $?B_1^-, \ldots, ?B_n^-$. The **translation of a** pn**-typed derivation** $\pi = \Gamma \vdash_{\mathsf{pn}} t : A$ **into a MELL proof-net**, written $\pi^\bullet$, is defined as the multiplicative normal form (*i.e.* the $\mathtt{C}(a)/\mathtt{C}(⅋, \otimes)$-normal form) of the PN π°, where π° is a MELL PN with conclusions $?\Gamma^-, A^+$ constructed by induction on π as follows:

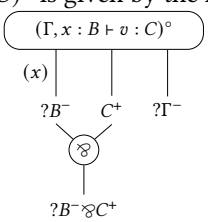
- If t is a variable x, then $(x : A \vdash_{\mathsf{pn}} x : A)^\circ$ is the following MELL proof-net



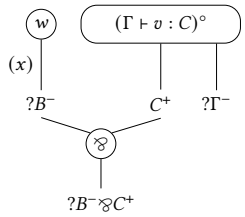
- If t is an application vu, then $(\Delta_v, \Delta_u, \Delta_{vu} \vdash_{\mathsf{pn}} vu : A)^\circ$, where $\mathsf{supp}(\Delta_v) = \mathtt{fv}(v) \setminus \mathtt{fv}(u)$, $\mathsf{supp}(\Delta_u) = \mathtt{fv}(u) \setminus \mathtt{fv}(v)$, and $\mathsf{supp}(\Delta_{vu}) = \mathtt{fv}(v) \cap \mathtt{fv}(u)$, is given by the following MELL proof-net



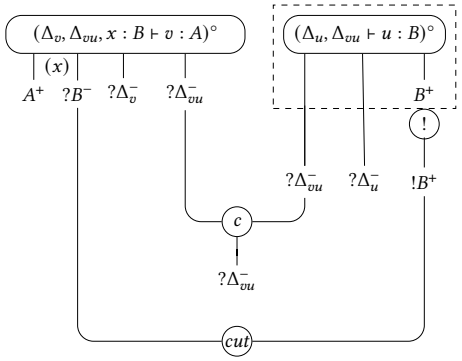
  Each wire in $?\Delta_{vu}^-$ is labelled with its corresponding variable in $\mathtt{fv}(v) \cap \mathtt{fv}(u)$.
- If t is an abstraction $\lambda x.v$, there are two cases:
  – If $x \in \mathtt{fv}(v)$, then $(\Gamma \vdash \lambda x.v : B \Rightarrow C)^\circ$ is given by the following MELL proof-net



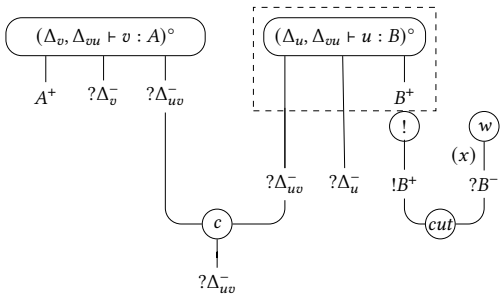
  – $x \notin \mathtt{fv}(v)$, then $(\Gamma \vdash \lambda x.v : B \Rightarrow C)^\circ$ is given by the following MELL proof-net



- If t is a closure $v[x\backslash u]$ there are again two cases:
  - If $x \in \mathtt{fv}(v)$, then $(\Delta_v, \Delta_u, \Delta_{vu} \vdash v[x\backslash u] : A)^\circ$, where $\mathsf{supp}(\Delta_v) = \mathtt{fv}(v) \setminus \{x\} \setminus \mathtt{fv}(u)$, $\mathsf{supp}(\Delta_u) = \mathtt{fv}(u) \setminus \mathtt{fv}(v)$, and $\mathsf{supp}(\Delta_{vu}) = \mathtt{fv}(v) \cap \mathtt{fv}(u)$ is given by the following MELL proof-net



    Each wire in $?\Delta^-_{vu}$ is labelled with its corresponding variable in $\mathtt{fv}(v) \cap \mathtt{fv}(u)$.
  - If $x \notin \mathtt{fv}(v)$, then $(\Delta_v, \Delta_u, \Delta_{vu} \vdash_{\mathsf{pn}} v[x\backslash u] : A)^\circ$, where where $\mathsf{supp}(\Delta_v) = \mathtt{fv}(v) \setminus \{x\} \setminus \mathtt{fv}(u)$, $\mathsf{supp}(\Delta_u) = \mathtt{fv}(u) \setminus \mathtt{fv}(v)$, and $\mathsf{supp}(\Delta_{vu}) = \mathtt{fv}(v) \cap \mathtt{fv}(u)$ is given by the following MELL proof-net



    Each wire in $?\Delta^-_{vu}$ is labelled with its corresponding variable in $\mathtt{fv}(v) \cap \mathtt{fv}(u)$.

Remark that even when $(\Gamma \vdash_{\mathsf{pn}} t : A)^\circ$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet$ are different, they have the same box structure, in the sense that $\_^\bullet$ does not add or remove boxes, nor does it modify the nesting of boxes. Since $\simeq_{\mathcal{E}}$-equivalence on proof-nets also preserves the box structure, then $\simeq_{\mathcal{E}}$-equivalent proof-nets in multiplicative normal form have the same box structure too. More interestingly, equivalent λ-terms in the sense of Regnier $\_^\bullet$ translate to structural equivalent proof-nets, because they have the same box structure. However, they do not $\_^\circ$ translate to structural equivalent proof-nets, because they do not necessarily have the same multiplicative structure, as illustrated by the terms $(\lambda x.(\lambda y.t)u)v$ and $(\lambda x.\lambda y.t)vu$ in the Strong Bisimulation subsection. Since our notion of $\simeq_\sigma$-equivalence on pn-terms is obtained from Regnier's equivalence on λ-terms by just removing some multiplicative cuts on the level of terms, then it is not surprising that not only the translation $\_^\bullet$, but also $\_^\circ$, realizes a structural equivalence on $\simeq_\sigma$-equivalent terms:

Lemma 6.1. *Let $\pi_1 = \Gamma \vdash_{\mathsf{pn}} t_1 : A$. If $t_1 \simeq_\sigma t_2$, then there exists $\pi_2 = \Gamma \vdash t_2 : A$ such that both $\pi_1^\circ \simeq_\mathcal{E} \pi_2^\circ$ and $\pi_1^\bullet \simeq_\mathcal{E} \pi_2^\bullet$ hold.*

Proof. The proof proceeds by induction on $\simeq_\sigma$. In the base case $t_1 = \mathsf{H}\langle s\rangle[x\backslash u] \simeq_\sigma \mathsf{H}\langle s[x\backslash u]\rangle = t_2$ with $x \notin \mathtt{fv}(\mathsf{H})$ and $\mathtt{fc}(u, \mathsf{H})$, we proceed by induction on H. The base case $\mathsf{H} = \Box$ is straightforward. The inductive cases are easy because head contexts do not contain any box. Only the equation $\equiv_{\mathsf{E}(c,c)}$ could be necessary to conclude when H and u share some common free variables. □

Soundness and Completeness. The static translation from typable pn-terms to MELL proof-nets guarantees that the congruence $\simeq_\sigma$ relates terms corresponding to (structural) equivalent proof-nets (*c.f.* Lem. 6.1), but it is worth noticing that the converse does not hold. Indeed, there exist λ-terms t_1 and t_2 translating to the same proof-net, which are not $\simeq_\sigma$-equivalent, as *e.g.* $t_1 := (\lambda x.(\lambda y.y)w)z$ and $t_2 := (\lambda x.(\lambda y.y))zw$. Indeed, the terms t_1 and t_2 are equivalent in the sense of Regnier [Regnier 1994] (so in particular they $\_^\bullet$ translate to structural equivalent proof-nets), but not in the sense of our equivalence relation $\simeq_\sigma$ on pn-terms, particularly because $\simeq_\sigma$ only relates terms with explicit substitutions, and not pure terms. This mismatch can be repaired by defining the following **structural equivalence** $\cong$ on pn-terms:

$$t \cong u \text{ iff } t(\simeq_{\mathsf{dB}} \cup \simeq_\sigma)u$$

where $\simeq_{\mathsf{dB}}$ is the reflexive, symmetric and transitive congruence generated by the distant Beta rule $\mapsto_{\mathsf{dB}}$ introduced in Sec. 2. Indeed, it is possible to show that two λ-terms t_1 and t_2 are equivalent in the sense of Regnier if and only if $t_1 \cong t_2$. Thus *e.g.* $t_1 \simeq_{\mathsf{dB}} y[y\backslash w][x\backslash z] \simeq_{\mathsf{dB}} t_2$. Now, structural equivalence on pn-terms $\_^\bullet$ translates (but not $\_^\circ$ translates) to structural equivalence on MELL PN:

Theorem 6.2 (Soundness). *Let $\pi_1 = \Gamma_1 \vdash_{\mathsf{pn}} t_1 : A_1$. If $t_1 \cong t_2$, then there exists $\pi_2 = \Gamma_2 \vdash_{\mathsf{pn}} t_2 : A_2$ such that $\pi_1^\bullet \simeq_\mathcal{E} \pi_2^\bullet$.*

We now show completeness, *i.e.* the structural equivalence $\simeq_\mathcal{E}$ on proof-nets corresponds to the structural equivalence $\cong$ on pn-terms. For that, we alternatively interpret a proof-net as a finite acyclic oriented graph, where the nodes are labeled by the following alphabet $\{cut, ax, \parr, \otimes, c, w, d, b\}$. Each node has a number of `in` and `out` ports. Indeed, the *cut*-node has two ports `in`, the *ax*-node has two ports `out`, the $\parr$, $\otimes$ and *c*-nodes have two ports `in` and one port `out`, the *w*-node has one port `out`, the *d*-node has one port `in` and one port `out`, and the *b*-node has *n* ports `in` and *n* ports `out` for $n \geq 1$. Each edge is decorated with a type and is connected to exactly one port `out` and at most one port `in`, i.e. each edge has a source node but not necessary a target node.

We say that a node *n* occurs at depth *d* in a proof net π iff *n* appears inside *d* different boxes of π. We say that a node *n* is **pre-final** in a proof net π iff there exists an edge connected to an `out` port of *n* but not to an `in` port of another node of π. Thus, pre-final nodes represent the **interface** of the proof-net as their outgoing edges are free. Remark that the *cut*-node is never pre-final. In order to adapt these notions to the relation $\simeq_\mathcal{E}$ on proof-nets, we finally define a node *n* to be **final** in π, if *n* is pre-final in the proof-net π' resulting from π by application of the equations $\mathsf{E}(b, c)$ and $\mathsf{E}(b, w)$. Given a proof-net π and a final node *f* of π we define $\pi \setminus f$ as the proof-net π **deprived** from node *n* and its corresponding edges. This operation is simply used to inductively reason on the number of nodes of proof-nets.

Remark that given two MELL proof-nets *p* and *q* such that $p \simeq_\mathcal{E} q$ their final nodes are the same. As a consequence, if $\pi_1 = \Gamma_1 \vdash_{\mathsf{pn}} t_1 : A_1$, $\pi_2 = \Gamma_2 \vdash_{\mathsf{pn}} t_2 : A_2$ and $t_1 \cong t_2$, then the final nodes of $\pi_1^\bullet$ and $\pi_2^\bullet$ are the same.

We start the completeness proof by some auxiliary lemmas.

Lemma 6.3. *Let $\pi = \Gamma \vdash_{\mathsf{pn}} t : A$ and f be a final $\parr$-node of $\pi^\bullet$. Then there exists y, u' such that $\lambda y.u' \cong t$.*

Proof. The proof is based on the fact that for every pn-term t, there is $u \cong t$ such that u is either a λ-abstraction or has the form $(xt_1 \dots t_n)[x_1 \backslash u_1] \dots [x_m \backslash u_m]$, where $n, m \geq 0$. □

Lemma 6.4. *Let $\pi = \Gamma \vdash_{\mathsf{pn}} t : A$. If $\pi^\bullet$ has no $\parr$-node but some cut-node at depth 0, then there exist u, v, x such that $t \cong (\lambda x.u)v$.*

Proof. The proof proceed by case analysis. □

We can now conclude:

Theorem 6.5 (Completeness). *Let $\pi_1 = \Gamma \vdash_{\mathsf{pn}} t_1 : A$ and $\pi_2 = \Gamma \vdash_{\mathsf{pn}} t_2 : A$. If ${\pi_1}^\bullet \simeq_{\mathcal{E}} {\pi_2}^\bullet$, then $t_1 \cong t_2$.*

Proof. By induction on the size (*i.e.* number of nodes) of ${\pi_1}^\bullet$.

- If $\pi_1^\bullet$ has a final $\parr$-node (let us say n), then $\pi_2^\bullet$ also does. Therefore $t_1 \cong \lambda y.t_1'$ and $t_2 \cong \lambda y.t_2'$ by Lem. 6.3. By definition of the translation we then have $A = B \Rightarrow C$, where $(\Gamma; y : B \vdash_{\mathsf{pn}} t_1' : C)^\bullet = (\Gamma \vdash_{\mathsf{pn}} t_1 : A)^\bullet \setminus n$ and $(\Gamma; y : B \vdash_{\mathsf{pn}} t_2' : C)^\bullet = (\Gamma \vdash_{\mathsf{pn}} t_2 : C)^\bullet \setminus n$. Thus $(\Gamma; y : B \vdash_{\mathsf{pn}} t_1' : C)^\bullet \simeq_{\mathcal{E}} (\Gamma; y : B \vdash_{\mathsf{pn}} t_2' : C)^\bullet$. By the *i.h.* we have $t_1' \cong t_2'$ so that $t_1 \cong \lambda y.t_1' \cong \lambda y.t_2' \cong t_2$.
- If $\pi_1^\bullet$ has no final $\parr$-node, then t_1 is necessarily an application, a variable or a closure. There are two cases to consider:
  - If $\pi_1^\bullet$ has no *cut*-node at depth 0, then $t_1 \cong xu_1 \dots u_n$ and $t_2 \cong xv_1 \dots v_n$ for $n \geq 0$, where the subterms are typed by derivations $\pi_{1,i} = \Delta^i \vdash_{\mathsf{pn}} u_i : C_i$ and $\pi_{2,i} = \Delta^i \vdash_{\mathsf{pn}} v_i : C_i$ for $i = 1 \dots n$. Moreover, ${\pi_{1,i}}^\bullet \simeq_{\mathcal{E}} {\pi_{2,i}}^\bullet$ and $\pi_{j,i}^\bullet$ is smaller than $\pi_j^\bullet$ for $j = 1, 2$. The *i.h.* gives $u_i \cong v_i$ $(i = 1 \dots n)$. Then $t_1 \cong xu_1 \dots u_n \cong xv_1 \dots v_n \cong t_2$.
  - If $\pi_1^\bullet$ has some *cut*-node at depth 0, then by Lem. 6.4 there exists u_1, v_1, x_1 such that $t_1 \cong (\lambda x_1.u_1)v_1$. The same happens with $\pi_2^\bullet$, so that Lem. 6.4 gives u_2, v_2, x_2 such that $t_2 \cong (\lambda x_2.u_2)v_2$. The proof-net $\pi_1^\bullet$ has two sub proof-nets $\pi_{1,1}^\bullet$ and $\pi_{1,2}^\bullet$ coming resp. from $\pi_{1,1} = \Delta_1 \vdash_{\mathsf{pn}} \lambda x_1.u_1 : B \Rightarrow C$ and $\pi_{1,2} = \Delta_2 \vdash_{\mathsf{pn}} v_1 : B$. Moreover, $\pi_{1,1}^\bullet$ and $\pi_{1,2}^\bullet$ are smaller than $\pi_1^\bullet$. The same happens with $\pi_2^\bullet$. Therefore $\lambda x_1.u_1 \cong \lambda x_2.u_2$ and $v_1 \cong v_2$ which imply $t_1 \cong (\lambda x_1.u_1)v_1 \cong (\lambda x_2.u_2)v_2 \cong t_2$.

□

6.2 The Dynamic Translation

This section states and proves a fine-grained relation between reduction on terms and reduction on MELL proof-nets. More precisely, we show that $\_^\bullet$ is injective, as every *single* exponential step on terms is captured by a *single* exponential step on PN. Thus, the neat discrimination between "boxing" and "duplication" holds also on the level of terms.

Given an environment $\Delta = x_1 : A_1, \dots, x_n : A_n$ $(n \geq 0)$ and a proof-net p, we write $\mathcal{W}_\Delta[p]$ for the proof-net obtained by adding n weakening wires, labeled respectively with the formulae $?\mathsf{A}_1^- \dots ?\mathsf{A}_n^-$, on the top level of p (outside all the boxes). The following property is the key result of our dynamic translation, it justifies in particular the use of the additional equations $\mathcal{E}$ in Fig. 5.

Theorem 6.6 (Faithful Dynamic Translation). *Let $\Gamma \vdash_{\mathsf{pn}} t : A$.*

- *If $t \to_{\mathsf{dB}} t'$, then $\Gamma \vdash_{\mathsf{pn}} t' : A$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet \simeq_{\mathcal{E}} (\Gamma \vdash_{\mathsf{pn}} t' : A)^\bullet$.*
- *If $t \to_{\mathsf{hs}} t'$, then $\Gamma \vdash_{\mathsf{pn}} t' : A$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet \to_{\mathsf{C}(d,b)/\mathcal{E}} (\Gamma \vdash_{\mathsf{pn}} t' : A)^\bullet$.*
- *If $t \to_{\mathsf{arg}} t'$, then $\Gamma \vdash_{\mathsf{pn}} t' : A$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet \to_{\mathsf{C}(b,b)/\mathcal{E}} (\Gamma \vdash_{\mathsf{pn}} t' : A)^\bullet$.*
- *If $t \to_{\mathsf{dup}} t'$, then $\Gamma \vdash_{\mathsf{pn}} t' : A$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet \to_{\mathsf{C}(c,b)/\mathcal{E}} (\Gamma \vdash_{\mathsf{pn}} t' : A)^\bullet$.*
- *If $t \to_{\mathsf{gc}} t'$, then $\Gamma' \vdash_{\mathsf{pn}} t' : A$ for $\Gamma' \subseteq \Gamma$ and $(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet \to_{\mathsf{C}(w,b)/\mathcal{E}} \mathcal{W}_{\Gamma \setminus \Gamma'}[(\Gamma' \vdash_{\mathsf{pn}} t' : A)^\bullet]$.*

Proof. The proof proceeds by induction on the reduction relations, *i.e.* by induction on the context $\mathtt{C}$ where the rewriting rule is applied. In all the base cases (*i.e.* $\mathtt{C} = \Box$) the $\mathtt{L}$ and $\mathtt{H}$ contexts can be omitted by just using $\simeq_\sigma$ equivalence and Lem. 6.1. Thus for example $\mathtt{H}\langle\langle x\rangle\rangle[x\backslash u] \to_{\mathtt{hs}} \mathtt{H}\langle\langle u\rangle\rangle$ can be decomposed as $\mathtt{H}\langle\langle x\rangle\rangle[x\backslash u] \simeq_\sigma \mathtt{H}\langle\langle x[x\backslash u]\rangle\rangle \to_{\mathtt{hs}} \mathtt{H}\langle\langle u\rangle\rangle$, so thanks to Lem. 6.1 it is sufficient to consider only $\mathtt{hs}$-reduction steps of the form $x[x\backslash u] \to_{\mathtt{hs}} u$. The same observation applies for all the other cases. More precisely,

- If $t = (\lambda x.s)\ u \to_{\mathtt{dB}} s[x\backslash u] = t'$, then $(\Gamma \vdash_{\mathtt{pn}} t : A)^\circ \to_{\mathtt{C}(a)/\mathtt{C}(\parr,\otimes)}$-reduces to $(\Gamma \vdash_{\mathtt{pn}} t' : A)^\circ$ so that $(\Gamma \vdash_{\mathtt{pn}} t : A)^\bullet \simeq_{\mathcal{E}} (\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$.
- If $t = x[x\backslash u] \to_{\mathtt{hs}} u = t'$, then the result is straightforward.
- If $t = \mathtt{A}\langle v\rangle[x\backslash u] \to_{\mathtt{arg}} \mathtt{A}\langle v[x\backslash u]\rangle = t'$, then $(\Gamma \vdash_{\mathtt{pn}} t : A)^\bullet \to_{\mathtt{C}(b,b)}$-reduces to another proof-net which is $\sim_{\mathcal{E}}$-equivalent (by means of $\equiv_{\mathtt{E}(b,c)}$) to $(\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$.
- If $t = \mathtt{A}\langle u\rangle[x\backslash v] \to_{\mathtt{dup}} \mathtt{A}_{[x\backslash v]}\langle u\rangle[x\backslash v] = t'$, then $(\Gamma \vdash_{\mathtt{pn}} t : A)^\bullet \to_{\mathtt{C}(c,b)}$-reduces to another proof-net which is $\sim_{\mathcal{E}}$-equivalent (by means of $\equiv_{\mathtt{E}(c,c)}$) to $(\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$.
- $t = s[x\backslash u] \to_{\mathtt{gc}} s = t'$, with $x \notin \mathtt{fv}(s)$, then $(\Gamma \vdash_{\mathtt{pn}} t : A)^\bullet \to^*_{\mathtt{w\text{-}b}} \simeq_{\mathcal{E}} \mathcal{W}_{\Gamma\backslash\Gamma'}[(\Gamma' \vdash_{\mathtt{pn}} t' : A)^\bullet]$, where $\Gamma' \subseteq \Gamma$ and $\mathcal{W}_{\Gamma\backslash\Gamma'}$ denotes all the weakening wires labelled with variables in Γ but not in Γ'. Moreover, equation $\equiv_{\mathtt{E}(c,w)}$ could be necessary to conclude when s and u share some free variables.

We now consider the cases where $t \to_{\mathtt{pn}} t'$ is an internal reduction step.

- If $t \simeq_\sigma t'$ then the property trivially holds since $\simeq_{\mathcal{E}}$ is a congruence.
- If $t \to t'$ is $\lambda x.s \to \lambda x.s'$ or $s\ u \to s'\ u$ or $s[x\backslash u] \to s'[x\backslash u]$ coming from $s \to s'$, then we obtain $(\Delta \vdash_{\mathtt{pn}} s : B)^\bullet \to^+_{\mathcal{R}/\mathcal{E}} \mathcal{W}[(\Delta \vdash_{\mathtt{pn}} s' : B)^\bullet]$ by the *i.h.* and the property holds by the fact that the context $\mathcal{W}[\ ]$ of weakening wires surrounding $(\Delta \vdash_{\mathtt{pn}} s' : B)^\bullet$ can also be considered as a context of weakening wires surrounding $(\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$.
- If $t \to t'$ is $u\ s \to u\ s'$ or $u[x\backslash s] \to u[x\backslash s']$ coming from $s \to s'$, then we obtain $(\Delta \vdash_{\mathtt{pn}} s : B)^\bullet \to^+_{\mathcal{R}/\mathcal{E}} \mathcal{W}[(\Delta \vdash_{\mathtt{pn}} s' : B)^\bullet]$ by the *i.h.* and the property holds by the fact that the context $\mathcal{W}[\ ]$ of weakening wires surrounding $(\Delta \vdash_{\mathtt{pn}} s' : B)^\bullet$ can be pushed outside the box containing $(\Delta \vdash_{\mathtt{pn}} s' : B)^\bullet$ by using the equation $\equiv_{\mathtt{E}(b,w)}$ in order to obtain a context of weakening wires surrounding $(\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$.

□

The converse implication does not necessarily hold: thus *e.g.* if $(\Gamma \vdash_{\mathtt{pn}} t : A)^\bullet \simeq_{\mathcal{E}} (\Gamma \vdash_{\mathtt{pn}} t' : A)^\bullet$ we can perfectly have $t \simeq_\sigma t'$ but not $t \to_{\mathtt{dB}} t'$. Remark also that the last case of the statement is the only one adding weakening wires on the top level of the translation of t': this is because $\mathtt{gc}$ is the only rule which eventually loses free variables (*c.f.* Lem. 2.1).

Notice also that every $\mathtt{dB}$-step on the term side is captured by the computation of the multiplicative-normal form used to construct the proof-net $\_^\bullet$ (translation of term type derivations into MELL PN). In addition, every single step in $\{\mathtt{gc}, \mathtt{hs}, \mathtt{arg}, \mathtt{dup}\}$ is captured by a single exponential step on the proof-net side. This phenomenon perfectly explains why we call multiplicative (resp. exponential) the $\mathtt{dB}$-step (resp. any step in $\{\mathtt{gc}, \mathtt{hs}, \mathtt{arg}, \mathtt{dup}\}$) on the terms. The analogy between steps on terms and proof-nets can be summarized by the following table:

pn-rewriting rule	MELL PN cut elimination rule	Meaning
$\mathtt{dB}$	$\mathtt{C}(a)/\mathtt{C}(\parr,\otimes)$	Application becomes ES
$\mathtt{gc}$	$\mathtt{C}(w,b)$	Erasure
$\mathtt{hs}$	$\mathtt{C}(d,b)$	Linear Substitution
$\mathtt{arg}$	$\mathtt{C}(b,b)$	Commutative Step
$\mathtt{dup}$	$\mathtt{C}(c,b)$	Duplication

It is worth mentioning that the dynamic simulation does not only hold for the $\_^\bullet$ translation but also for the $\_^\circ$ translation: in particular, $\to_{\mathsf{dB}}$ on pn-terms simulates with $\mathsf{C}(⅋, ⊗)$ and $\mathsf{C}(a)$ on proof-nets, and $\to_{\mathsf{hs}}$ simulates with $\mathsf{C}(d, b)$ and $\mathsf{C}(a)$. Thus, the $\_^\circ$ translation is not injective, and the magic one-to-one simulation between reduction on terms and proof-nets disappears. That is why our simulation result is stated with the function $\_^\bullet$.

7 STRONG NORMALIZATION

We now show that all *typable* pn-terms are strong normalizing w.r.t. the reduction relation $\to_{\mathsf{pn}}$. Indeed, given any reduction relation $\to_{\mathcal{R}}$, a **term** t is said to be **$\mathcal{R}$-strongly normalizing**, written $t \in \mathcal{SN}_{\mathcal{R}}$, iff there is no infinite pn-reduction sequence starting at t; and the **reduction relation** $\mathcal{R}$ is **strongly normalizing (SN)**, if every term t verifies $t \in \mathcal{SN}_{\mathcal{R}}$. It is clear that $\mathcal{SN}_{\mathsf{pn}}$ cannot hold in general for all terms, because strong-normalization does not hold for $\to_\beta$, and $\to_\beta$ *strictly* translates to $\to^+_{\mathsf{pn}}$ (Lem. 3.2). Strong normalization for typable terms can be shown by resorting to the dynamic translation presented in Section 6.2 and the following abstract rewriting theorem.

Theorem 7.1. *Let O, P be two sets. Let $\mathcal{R}, \mathcal{S}$ be two relations on $\mathsf{O} \times \mathsf{O}$, $\mathcal{U}$ be a relation on $\mathsf{P} \times \mathsf{P}$, and $\mathcal{K}$ a relation on $\mathsf{O} \times \mathsf{P}$. Let us denote by $\mathcal{U}^*$ (resp. $\mathcal{U}^+$) the reflexive-transitive (resp. transitive) closure of $\mathcal{U}$. Assume also:*

(1) *$\mathcal{R}$ is SN*
(2) *$t\ \mathcal{R}\ t'$ and $t\ \mathcal{K}\ T$ implies there is $T' \in \mathsf{P}$ such that $t'\ \mathcal{K}\ T'$ and $T\ \mathcal{U}^*\ T'$*
(3) *$t\ \mathcal{S}\ t'$ and $t\ \mathcal{K}\ T$ implies there is $T' \in \mathsf{P}$ such that $t'\ \mathcal{K}\ T'$ and $T\ \mathcal{U}^+\ T'$*

Then $t\ \mathcal{K}\ T$ and $T \in \mathcal{SN}_{\mathcal{U}}$ imply $t \in \mathcal{SN}_{\mathcal{R}\cup\mathcal{S}}$.

Proof. Suppose $t\ \mathcal{K}\ T$, $T \in \mathcal{SN}_{\mathcal{U}}$ and $t \notin \mathcal{SN}_{\mathcal{R}\cup\mathcal{S}}$. Since $\mathcal{R}$ is SN by **(1)**, then there is an infinite sequence on O where $\mathcal{S}$ occurs infinitely many times:

$$t = t_0\ \ldots \mathcal{R}^* \mathcal{S}\ t_1 \ldots \mathcal{R}^* \mathcal{S}\ t_2 \ldots \mathcal{R}^* \mathcal{S}\ t_i \ldots$$

But $t\ \mathcal{K}\ T$, then by **(2)** and **(3)** there are $T_1, T_2, \ldots, T_i, \ldots \in \mathsf{P}$ such that $t_1\ \mathcal{K}\ T_1, t_2\ \mathcal{K}\ T_2, \ldots, t_i\ \mathcal{K}\ T_i, \ldots$ and the following infinite $\mathcal{U}$-reduction sequence is then generated

$$T\ \mathcal{U}^+\ T_1\ \mathcal{U}^+\ T_2\ \mathcal{U}^+\ \ldots \mathcal{U}^+\ T_i\ \ldots$$

This leads to a contradiction with the hypothesis $T \in \mathcal{SN}_{\mathcal{U}}$. □

Corollary 7.2. *The reduction relation $\to_{\mathsf{pn}}$ is strongly normalizing on typable terms.*

Proof. We let O be the set of typable pn-terms, $\mathcal{R} := \to_{\mathsf{dB}}$, and $\mathcal{S} := \to_{\mathsf{gc,hs,arg,dup}}$. Let also consider P be the set of MELL proof-nets, $\mathcal{U} := \to_{\mathcal{R}/\mathcal{E}}$, and $\mathcal{K}$ the relation given by $t\mathcal{K}\mathcal{W}_\Delta[(\Gamma \vdash_{\mathsf{pn}} t : A)^\bullet]$ for any environment Δ. The property then holds by the abstract Theorem 7.1 instantiated to the sets and relations described above. Indeed, **(2)** and **(3)** hold by Thm. 6.6, $\to_{\mathsf{dB}}$ is straightforwardly SN, and $\to_{\mathcal{R}/\mathcal{E}}$ is SN on MELL proof-nets by [Di Cosmo and Piperno 1995]. □

The previous result can be extended to a stronger property stating that typable pn-terms are strongly normalizing w.r.t. the pn-**reduction modulo σ-equivalence**, where $t \to_{\mathsf{pn}/\sigma} t'$ iff there are u, u' such that $t \simeq_\sigma t' \to_{\mathsf{pn}} u' \simeq_\sigma u$. We omit here the details by lack of space.

8 PRESERVATION OF STRONG NORMALIZATION

Preservation of strong normalization (PSN) is a property about *untyped* terms, that becomes crucial when implementing a source language $\mathcal{S}$ with a more refined target calculus $\mathcal{T}$ (possibly extending the syntax of $\mathcal{S}$), simply because $\mathcal{T}$ is expected to preserve the underlying properties of $\mathcal{S}$, including termination. Indeed, we say that $\mathcal{T}$ **preserves $\mathcal{S}$-strong normalization**, or $\mathcal{T}$ **enjoys $\mathcal{S}$-PSN**, if

$t \in \mathcal{SN}_{\mathcal{S}}$ implies $t \in \mathcal{SN}_{\mathcal{T}}$ for every term t. PSN received a lot of attention in the theory of calculi with ES, starting from an unexpected result by Melliès [Melliès 1995], who showed that $\lambda\sigma$ [Abadi et al. 1991] does not enjoy β-PSN. When PSN holds, it is usually a non-trivial property to prove. Here we adopt a technique projecting $\to_{\mathsf{pn}}$ into another reduction relation enjoying β-PSN itself. For that, we consider the structural lambda-calculus $\lambda\mathtt{j}$ modulo an equivalence relation [Accattoli and Kesner 2012], a choice that could be perceived as surprising, since $\lambda\mathtt{j}$ is less fine-grained than pn. However, the simulation proof (Lem. 8.2) perfectly works because the target calculus is considered *modulo* a set of equations $\equiv_{\mathsf{obox}}$, which encompasses in particular the commutative rules hidden in Girard's MELL proof-nets, and thus those in pn.

The **reduction relation** $\to_{\lambda\mathtt{j}}$ is given by the closure of all contexts of the four rewriting rules:

$$\begin{array}{llll} \mathtt{L}\langle\lambda x.t\rangle u & \mapsto_{\mathsf{dB}} & \mathtt{L}\langle t[x\backslash u]\rangle & \\ t[x\backslash u] & \mapsto_{\mathsf{w}} & t & \text{if } |t|_x = 0 \\ t[x\backslash u] & \mapsto_{\mathsf{d}} & t\{x\backslash u\} & \text{if } |t|_x = 1 \\ t[x\backslash u] & \mapsto_{\mathsf{c}} & t_{[y]_x}[x\backslash u][y\backslash u] & \text{if } |t|_x > 1 \end{array}$$

where rule dB is exactly the same used in the pn-calculus, and $t_{[y]_x}$ denotes any **non-deterministic replacement** of i $(1 \le i \le n-1)$ occurrences of x in t by a *fresh* variable y, an operation which is always defined when $|t|_x > 1$. Here is an example: $(\lambda x.xxx)z \to_{\mathsf{dB}} (xxx)[x\backslash z] \to_{\mathsf{c}} (yxy)[x\backslash z][y\backslash z] \to_{\mathsf{d}} (yzy)[y\backslash z] \to_{\mathsf{c}} (yzy')[y\backslash z][y'\backslash z] \to_{\mathsf{d}} (yzz)[y\backslash z] \to_{\mathsf{d}} zzz$.

The equivalence relation $\equiv_{\mathsf{obox}}$ is the reflexive, symmetric, transitive and closed by all contexts relation generated by the following axioms:

$$\begin{array}{lcll} t[y\backslash v][x\backslash u] & \simeq & t[x\backslash u][y\backslash v] & \text{if } x \notin \mathtt{fv}(v), y \notin \mathtt{fv}(u) \\ t[y\backslash v][x\backslash u] & \simeq & t[y\backslash v[x\backslash u]] & \text{if } x \notin \mathtt{fv}(t), x \notin \mathtt{fv}(v) \\ (\lambda y.t)[x\backslash u] & \simeq & \lambda y.t[x\backslash u] & \text{if } y \notin \mathtt{fv}(u) \\ (tv)[x\backslash u] & \simeq & t[x\backslash u]v & \text{if } x \notin \mathtt{fv}(v) \\ (tv)[x\backslash u] & \simeq & tv[x\backslash u] & \text{if } x \notin \mathtt{fv}(t), x \in \mathtt{fv}(v) \end{array}$$

We now define $\lambda\mathtt{j}$**-reduction modulo** obox**-equivalence**, as the relation such that $t \to_{\lambda\mathtt{j}/\mathsf{obox}} t'$ iff there are u, u' such that $t \equiv_{\mathsf{obox}} t' \to_{\lambda\mathtt{j}} u' \equiv_{\mathsf{obox}} u$. We based our result on the following result [Accattoli and Kesner 2012, Thm. 5.18]:

Theorem 8.1 (β-PSN for $\to_{\lambda\mathtt{j}/\mathsf{obox}}$). *The relation $\to_{\lambda\mathtt{j}/\mathsf{obox}}$ enjoys β-PSN.*

In order to show that $\to_{\mathsf{pn}}$ enjoys in turn β-PSN, we study the relation between $\to_{\mathsf{pn}}$ and $\to_{\lambda\mathtt{j}/\mathsf{obox}}$.

Lemma 8.2. *Let t be a* pn*-term.*

- *If $t \simeq_\sigma t'$, then $t \equiv_{\mathsf{obox}} t'$.*
- *If $t \to_{\mathtt{arg}} t'$, then $t \equiv_{\mathsf{obox}} t'$.*
- *If $t \to_{\mathsf{dB,gc,hs,dup}} t'$, then $t \to_{\lambda\mathtt{j}/\mathsf{obox}} t'$.*

Proof. We prove each item as follows:

- If $t \simeq_\sigma t'$, then $t \equiv_{\mathsf{obox}} t'$ is straightforward since $\simeq_\sigma$ is included in $\equiv_{\mathsf{obox}}$.
- If $t = \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle\rangle[x\backslash u]\rangle \to_{\mathtt{arg}} \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0[x\backslash u]\rangle\rangle\rangle = t'$, with $x \notin \mathtt{fv}(\mathtt{H}), x \in \mathtt{fv}(t_0), x \notin \mathtt{fv}(\mathtt{A})$, then $t = \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle\rangle[x\backslash u]\rangle \equiv_{\mathsf{obox}} \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0[x\backslash u]\rangle\rangle\rangle = t'$.
- If $t \to_{\mathsf{dB}} t'$ or $t \to_{\mathtt{gc}} t'$ this is straightforward. If $t = \mathtt{C}\langle\mathtt{H}\langle x\rangle[x\backslash u]\rangle \to_{\mathtt{hs}} \mathtt{C}\langle\mathtt{H}\langle u\rangle\rangle = t'$, with $x \notin \mathtt{fv}(\mathtt{H})$, then $t = \mathtt{C}\langle\mathtt{H}\langle x\rangle[x\backslash u]\rangle \to_{\mathtt{d}} \mathtt{C}\langle\mathtt{H}\langle x\rangle\{x\backslash u\}\rangle = \mathtt{C}\langle\mathtt{H}\langle u\rangle\rangle = t'$. Finally, if $t = \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle\rangle[x\backslash u]\rangle \to_{\mathtt{dup}} \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}_{[x\backslash u]}\langle t_0\rangle[x\backslash u]\rangle\rangle = t'$, with $x \notin \mathtt{fv}(\mathtt{H}), x \in \mathtt{fv}(t_0), x \in \mathtt{fv}(\mathtt{A})$, then $t = \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle\rangle[x\backslash u]\rangle \equiv_{\mathsf{obox}} \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle[x\backslash u]\rangle\rangle \to_{\mathtt{c}} \mathtt{C}\langle\mathtt{H}\langle\mathtt{A}\langle t_0\rangle[x\backslash u]\rangle\rangle t'$.

□

Corollary 8.3 (β-PSN for $\to_{\mathsf{pn}}$). *The relation $\to_{\mathsf{pn}}$ enjoys β-PSN.*

Proof. Let O and P be the set of (untyped) pn-terms, $\mathcal{R} := \to_{\mathsf{arg}}$, $\mathcal{S} := \to_{\mathsf{dB,gc,hs,dup}}$, $\mathcal{U} := \to_{\lambda\mathsf{j/obox}}$, and $\mathcal{K}$ the identity relation given by $t\mathcal{K}t$ for any pn-term t. The property holds by the abstract Theorem 7.1 instantiated to the sets and relations described above. Indeed, **(2)** and **(3)** hold by Lem. 8.2, $\to_{\mathsf{arg}}$ is straightforwardly SN, and t is $\to_{\lambda\mathsf{j/obox}}$-SN by Them. 8.1. □

The β-PSN property can be extended to pn-reduction modulo σ-equivalence, as we did for strong normalization in Sec. 7.

9 CONCLUSION

This paper bridges the gap between sequential syntaxes and graphical formalisms by proposing a term calculus which translates to Girard's MELL proof-nets with a faithful degree of granularity, both statically and dynamically. From a static point of view, we characterize equivalence relations on both sides (terms and PN) which are in perfect correspondence (Thm 6.2 and Thm 6.5). From a dynamic point of view, we show that the reduction relation of our calculus translates to Girard's cut elimination accurately, *i.e.* one-to-one (Thm. 6.6). Many non-trivial properties are established for the pn-calculus, in particular, confluence (Thm. 3.4), strong normalization (Cor. 7.2), and preservation of β-strong normalization (Thm. 8.3). We also define a strong bisimulation on pn-terms which captures Regnier's well-known σ-equivalence relation lifted to ES (Thm. 3.5), a property that only a few calculi with ES enjoy [Accattoli et al. 2014b].

Our work highlights the fact that Girard's implementation of substitution combines both induction on the structure of terms and induction on the number of free occurrences of variables, in contrast to all other existing implementations of substitution based entirely on one or the other. We argue that the pn-calculus can be used to introduce and explain linear logic from a Curry-Howard perspective, mainly its exponential fragment, which is the most intriguing.

While the main interest of the computational interpretation of the cut elimination process in linear logic proof-nets relies on its exponential fragment, it could be also intriguing to explore the case of other linear logic connectives. The multiplicative units for $\otimes$ and $⅋$ seems unproblematic, but it is however not clear how to extend the interpretation to the additive connectives [Hughes and van Glabbeek 2005], particularly to deal with the links for axioms on the level of variables.

As a future work, we would like to extend these ideas to polarized proof-nets [Laurent 2003] —an extension of MELL proof-nets for the polarized fragment of linear logic—, which would faithfully capture a functional programming language with continuations, probably materialized by some refined version of the $\lambda\mu$-calculus [Parigot 1992]. An alternative or complementary direction is to address the entire linear logic (without polarization), in the spirit of [Abramsky 1993], by keeping our interpretation of the exponential fragment.

Extending the model to real programming languages would need to consider different features such as data-types/pattern-matching, polymorphism, continuations and effects, etc. With that sight, an implementation scheme by means of an abstract machine is under consideration, it concerns in particular a mechanism to search the next redex to be reduced according to some concrete evaluation strategy (call-by-name, call-by-value, call-by-need, etc).

Alternative approaches to model cut elimination in linear logic use the geometry of interaction [Girard 1988a,b], where a flow of tokens around a network is used rather than graph-rewriting. It will be interesting to understand how our approach relates to this alternative view.

REFERENCES

Martín Abadi, Luca Cardelli, Pierre Louis Curien, and Jean-Jacques Lévy. 1991. Explicit substitutions. *Journal of Functional Programming* 4, 1 (1991), 375–416. https://doi.org/10.1017/S0956796800000186

Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical Computer Science* 111 (1993), 3–57. https://doi.org/10.1016/0304-3975(93)90181-R

Beniamino Accattoli. 2011. *Jumping around the box: graphical and operational studies on Lambda Calculus and Linear Logic.* Ph.D. Thesis. Università di Roma La Sapienza.

Beniamino Accattoli. 2018. Proof Nets and the Linear Substitution Calculus. In *15th Int. Colloquium on Theoretical Aspects of Computing (ICTAC) (Lecture Notes in Computer Science, Vol. 11187)*, Bernd Fischer and Tarmo Uustalu (Eds.). Springer, 37–61. https://doi.org/10.1007/978-3-030-02508-3_3

Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014a. Distilling Abstract Machines. In *19th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, Manuel Chakravarty (Ed.). ACM Press, 363–376. https://doi.org/10.1145/2628136.2628154

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

Beniamino Accattoli and Stefano Guerrini. 2009. Jumping Boxes. Representing lambda-calculus boxes by jumps. In *Proceedings of the 18th EACSL Conf. on Computer Science Logic (CSL) (Lecture Notes in Computer Science, Vol. 5771)*, Erich Grädel and Reinhard Kahle (Eds.). Springer-Verlag, 55–70. https://doi.org/10.1007/978-3-642-04027-6_7

Beniamino Accattoli and Delia Kesner. 2010. The Structural *lambda*-Calculus. In *Proceedings of 24th EACSL Conf. on Computer Science Logic (CSL) (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer-Verlag, 381–395. https://doi.org/10.1007/978-3-642-15205-4_30

Beniamino Accattoli and Delia Kesner. 2012. Preservation of Strong Normalisation modulo permutations for the structural λ-calculus. *Logical Methods in Computer Science* 8, 1 (2012), 1–44. https://doi.org/10.2168/LMCS-8(1:28)2012

Frances E. Allen (Ed.). 1990. . ACM Press.

Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. 1996. The Bologna Optimal Higher-Order Machine. *J. Funct. Program.* 6, 6 (1996), 763–810. https://doi.org/10.1017/S0956796800001994

Pablo Barenbaum and Eduardo Bonelli. 2017. Optimality and the Linear Substitution Calculus. In *2nd Int. Conf. on Formal Structures for Computation and Deduction (FSCD) (LIPIcs, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:16. https://doi.org/10.4230/LIPIcs.FSCD.2017.9

Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. 1993. A Term Calculus for Intuitionistic Linear Logic. In *Proceedings of the 1st Int. Conf. on Typed Lambda Calculus and Applications (TLCA)*, Jan Friso Groote and Marc Bezem (Eds.). Lecture Notes in Computer Science, Vol. 664. Springer-Verlag, 75–90. https://doi.org/10.1007/BFb0037099

Roel Bloo and Kristoffer Rose. 1995. Preservation of Strong Normalization in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *Computing Science in the Netherlands*. Netherlands Computer Science Research Foundation, 62–72.

Alonzo Church. 1932. A set of postulates for the foundation of logic. *Annals of Mathematics* 33, 2 (1932), 346–366.

Pierre-Louis Curien. 1991. An abstract framework for environment machines. *Theoretical Computer Science* 82, 2 (1991), 389–402. https://doi.org/10.1016/0304-3975(91)90230-Y

Vincent Danos. 1990. *La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du lambda-calcul).* Ph.D. Thesis. Université Paris 7.

Vincent Danos, Hugo Herbelin, and Laurent Regnier. 1996. Game Semantics & Abstract Machines. In *11th IEEE Symp. on Logic in Computer Science (LICS)*, Edmund M. Clarke (Ed.). IEEE Computer Society, 394–405. https://doi.org/10.1109/LICS.1996.561456

Vincent Danos and Laurent Regnier. 1999. Reversible, irreversible and optimal lambda-machines. *Theoretical Computer Science* 227, 1 (1999), 79–97. https://doi.org/10.1016/S0304-3975(99)00049-3

Olivier Danvy and Ian Zerny. 2013. A synthetic operational account of call-by-need evaluation. In *15th Int. Symp. on Principles and Practice of Declarative Programming, (PPDP)*, Ricardo Peña and Tom Schrijvers (Eds.). ACM Press, 97–108. https://doi.org/10.1145/2505879.2505898

Nicolaas Govert de Bruijn. 1978. *A namefree lambda calculus with facilities for internal definition of expressions and segments.* Technical Report 78-WSK-03. Eindhoven University of Technology, Eindhoven.

Nicolaas Govert de Bruijn. 1987. Generalizing Automath by Means of a Lambda-Typed Lambda Calculus. In *Mathematical Logic and Theoretical Computer Science (Lecture Notes in Pure and Applied Mathematics, 106)*, Edgar G.K. Lopez-Escobar David W. Kueker and Carl H. Smith (Eds.). Marcel Dekker, 71–92.

Mariangiola Dezani-Ciancaglini and Gordon Plotkin (Eds.). 1995. . Lecture Notes in Computer Science, Vol. 902. Springer-Verlag. https://doi.org/10.1007/BFb0014040

Roberto Di Cosmo and Delia Kesner. 1994. Simulating Expansions without Expansions. *Mathematical Structures in Computer Science* 4 (1994), 315–362. https://doi.org/10.1017/S0960129500000505

Roberto Di Cosmo and Delia Kesner. 1997. Strong Normalization of Explicit Substitutions via Cut Elimination in Proof Nets (Extended Abstract). In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, June 29 - July 2, 1997*. IEEE Computer Society, 35–46. https://doi.org/10.1109/LICS.1997.614927

Roberto Di Cosmo, Delia Kesner, and Emmanuel Polonovski. 2003. Proof Nets and Explicit Substitutions. *Mathematical Structures in Computer Science* 13, 3 (2003), 409–450. https://doi.org/10.1017/S0960129502003791

Roberto Di Cosmo and Adolfo Piperno. 1995. Expanding Extensional Polymorphism, See [Dezani-Ciancaglini and Plotkin 1995], 139–153. https://doi.org/10.1007/BFb0014050

Thomas Ehrhard and Laurent Regnier. 2006. Differential interaction nets. *Theoretical Computer Science* 364, 2 (2006), 166–195. https://doi.org/10.1016/j.tcs.2006.08.003

Maribel Fernández and Nikolaos Siafakas. 2014. Labelled calculi of resources. *J. Log. Comput.* 24, 3 (2014), 591–613. https://doi.org/10.1093/logcom/exs021

Neil Ghani, Valeria de Paiva, and Eike Ritter. 1999. Categorical Models of Explicit Substitutions *(Lecture Notes in Computer Science, Vol. 1578)*, Wolfgang Thomas (Ed.). Springer, 197–211. https://doi.org/10.1007/3-540-49019-1

Neil Ghani, Valeria de Paiva, and Eike Ritter. 2000. Linear Explicit Substitutions. *Logic Journal of the Interest Group of Pure and Applied Logic* 8, 1 (2000), 7–31. https://doi.org/10.1093/jigpal/8.1.7

Jean-Yves Girard. 1988a. Geometry of interaction 1: Interpretation of System F. In *Logic Colloquium 88*. North Holland.

Jean-Yves Girard. 1988b. Geometry of interaction 2: deadlock-free algorithms. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science, Vol. 417)*, Per Martin-Löf and Grigori Mints (Eds.). Springer, 76–93. https://doi.org/10.1007/3-540-52335-9_49

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard. 1996a. *Proof-nets: The parallel syntax for proof-theory*. Lecture Notes in Pure Applied Mathematics, Vol. 180. CRC Press, Boca Raton, FL, USA, 97–124.

Jean-Yves Girard. 1996b. Proof-nets: The parallel syntax for proof-theory. In *Logic and Algebra*. Marcel Dekker, 97–124.

Timothy Griffin. 1990. A Formulae-as-Types Notion of Control, See [Allen 1990], 47–58. https://doi.org/10.1145/96709.96714

Tom Gundersen, Willem Heijltjes, and Michel Parigot. 2013. Atomic Lambda Calculus: A Typed Lambda-Calculus with Explicit Sharing. In *28th Annual ACM/IEEE Symp. on Logic in Computer Science (LICS)*, Orna Kupferman (Ed.). IEEE Computer Society, 311–320. https://doi.org/10.1109/LICS.2013.37

Thérèse Hardin and Jean-Jacques Lévy. 1989. A Confluent Calculus of Substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*. 1–25.

Peter Henderson and James H. Morris. 1976. A Lazy Evaluator. In *3rd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL)*, Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman (Eds.). ACM Press, 95–103.

Hugo Herbelin. 1994. A λ-Calculus Structure Isomorphic to Gentzen-Style Sequent Calculus Structure. In *Proceedings of the 8th EACSL Conf. on Computer Science Logic (CSL) (Lecture Notes in Computer Science, Vol. 933)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer-Verlag, 61–75. https://doi.org/10.1007/BFb0022247

Dominic J. D. Hughes and Rob J. van Glabbeek. 2005. Proof nets for unit-free multiplicative-additive linear logic. *ACM Trans. Comput. Log.* 6, 4 (2005), 784–842. https://doi.org/10.1145/1094622.1094629

Fairouz Kamareddine and Alejandro Ríos. 1995. A λ-calculus à la de Bruijn with explicit substitutions. In *Proceedings of the 7th Int. Symp. on Proceedings of the International Symposium on Programming Language Implementation and Logic Programming (Lecture Notes in Computer Science, Vol. 982)*, Doaitse Swierstra and Manuel Hermenegildo (Eds.). Springer-Verlag, 45–62. https://doi.org/10.1007/BFb0026813

Delia Kesner. 2007. The Theory of Calculi with Explicit Substitutions Revisited. In *Proceedings of the 16th EACSL Conf. on Computer Science Logic (CSL) (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas Henzinger (Eds.). Springer-Verlag, 238–252. https://doi.org/10.1007/978-3-540-74915-8_20

Delia Kesner, Eduardo Bonelli, and Andrés Viso. 2020. Strong Bisimulation for Control Operators (Invited Talk). In *28th EACSL Annual Conf. on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain (LIPIcs, Vol. 152)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:23. https://doi.org/10.4230/LIPIcs.CSL.2020.4

Delia Kesner and Stéphane Lengrand. 2005. Extending the Explicit Substitution Paradigm. In *16th Int. Conf. on Rewriting Techniques and Applications (RTA) (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer-Verlag, 407–422. https://doi.org/10.1007/978-3-540-32033-3_30

Jean-Louis Krivine. 2007. A Call-by-name Lambda-calculus Machine. *Higher Order and Symbolic Computation* 20, 3 (Sept. 2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

Ambroise Lafont. 2019. *Signatures and models for syntax and operational semantics in the presence of variable binding.* Ph.D. Thesis. L'École Nationale Superieure Mines-Telecom Atlantique Bretagne Pays de la Loire - IMT Atlantique.

John Lamping. 1990. An algorithm for optimal lambda calculus reduction, See [Allen 1990], 16–30. https://doi.org/10.1145/96709.96711

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320.

Olivier Laurent. 2002. *Etude de la polarisation en logique.* Thèse de Doctorat. Université Aix-Marseille II.

Olivier Laurent. 2003. Polarized proof-nets and lambda-mu calculus. *Theoretical Computer Science* 1, 290 (2003), 161–188. https://doi.org/10.1016/S0304-3975(01)00297-3

Xavier Leroy. 1990. *The ZINC Experiment: an economical implementation of the ML language.* Technical Report 117. INRIA Rocquencourt.

Ian Mackie. 1998. YALE: Yet Another Lambda Evaluator Based on Interaction Nets. In *3rd ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM Press, 117–128. https://doi.org/10.1145/289423.289434

Per Martin-Löf. 1972. An intuitionistic theory of types. Unpublished.

Paul-André Melliès. 1995. Typed λ-calculi with explicit substitutions may not terminate, See [Dezani-Ciancaglini and Plotkin 1995], 328–334. https://doi.org/10.1007/BFb0014062

Robin Milner. 2006. Local bigraphs and confluence: two conjectures. In *Proceedings of the 13th Int. Workshop on Expressiveness in Concurrency (EXPRESS)*, Roberto Amadio and Iain Phillips (Eds.), Vol. 175. Electronic Notes in Theoretical Computer Science, 65–73. https://doi.org/10.1016/j.entcs.2006.07.035

Koko Muroya and Dan R. Ghica. 2017. The Dynamic Geometry of Interaction Machine: A Call-by-Need Graph Rewriter. In *26th EACSL Annual Conf. on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden (LIPIcs, Vol. 82)*, Valentin Goranko and Mads Dam (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:15. https://doi.org/10.4230/LIPIcs.CSL.2017.32

Robert. P. Nederpelt. 1992. *The fine-structure of lambda calculus.* Technical Report Computing Science Notes 92/07. Eindhoven University of Technology, Department of Mathematics and Computer Science.

Michel Parigot. 1992. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Int. Conf. on Logic Programming and Automated Reasoning (Lecture Notes in Computer Science, Vol. 624)*, Andrei Voronkov (Ed.). Springer-Verlag, 190–201. https://doi.org/10.1007/BFb0013061

Laurent Regnier. 1992. *Lambda-calcul et réseaux.* Ph.D. Thesis. Université Paris 7.

Laurent Regnier. 1994. Une équivalence sur les lambda-termes. *Theoretical Computer Science* 2, 126 (1994), 281–292. https://doi.org/10.1016/0304-3975(94)90012-4

Eike Ritter. 1999. Characterising Explicit Substitutions which Preserve Termination. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer-Verlag, 325–339. https://doi.org/10.1007/3-540-48959-2_23

Kristoffer Rose. 1992. Explicit Cyclic Substitutions. In *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems (CTRS) (Lecture Notes in Computer Science, Vol. 656)*, Michaël Rusinowitch and Jean-Luc Rémy (Eds.). Springer-Verlag, 36–50. https://doi.org/10.1007/3-540-56393-8_3

Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *J. Funct. Program.* 7, 3 (1997), 231–264.

Paula Severi and Erik Poll. 1994. Pure Type Systems with Definitions. In *Logical Foundations of Computer Science'94 (Lecture Notes in Computer Science, Vol. 813)*. Springer-Verlag, 316–328. https://doi.org/10.1007/3-540-58140-5_30

Terese. 2003. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science, Vol. 55. Cambridge University Press.

Christopher Wadsworth. 1971. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis. Oxford University.