Extending the explicit substitution paradigm

Delia Kesner[1] and Stéphane Lengrand[1,2]

[1] PPS, Université Paris 7, France {kesner,lengrand}@pps.jussieu.fr
[2] School of Computer Science, University of St Andrews, United Kingdom.

Abstract We present a simple term language with explicit operators for erasure, duplication and substitution enjoying a sound and complete correspondence with the intuitionistic fragment of Linear Logic's Proof Nets. We establish the good operational behaviour of the language by means of some fundamental properties such as confluence, preservation of strong normalisation, strong normalisation of well-typed terms and step by step simulation. This formalism is the first term calculus with explicit substitutions having full composition and preserving strong normalisation.

1 Introduction

The *Curry-Howard* paradigm, according to which the terms/types/reduction of a term language respectively correspond to the proofs/propositions/normalisation of a logical system, has already shown its numerous merits in the computer science community. Such a correspondence gives a double reading of proofs as programs and programs as proofs, so that insight into one aspect helps the understanding of the other.

A typical example of the Curry-Howard correspondence is obtained by taking the simply typed λ-calculus [11] as term language and Natural Deduction for Intuitionistic Logic as logical system. But both formalisms can be decomposed in the following sense: on the one hand the evaluation rule of λ-calculus, known as β-reduction, can be decomposed into more elementary operations: the various tasks needed to implement β can be achieved by manipulation of various *explicit operators*, such as *erasure, duplication* and *substitution*. On the other hand *Linear Logic* [24] decomposes the intuitionistic logical connectives into more elementary connectives, such as the linear arrow and the exponentials, thus providing a more refined use of resources than that of Intuitionistic Logic.

We show that there is a deep connection between these two elementary decompositions. In order to relate them, we must bridge the conceptual gap between the formalism of a term syntax and that of *Proof Nets* [24] that we use to denote proofs in Linear Logic. Visually convenient to manipulate, the latter retains from the structure of a proof the part that is logically relevant, thus giving geometric insight into proof transformations. However, it is quite cumbersome in proof formalisations. On the other hand, term notation is more convenient to formalise and carry detailed proofs of properties, and also when one wants to implement them via some proof-assistant [12,31].

Several works [16,17] have already explored the relation between these two approaches, but none of them has pushed the formalism far enough to obtain a computational counterpart to Proof Nets that is sound and complete with respect to the underlying logical model.

We present a calculus with explicit operators for erasure, duplication and substitution, called $\lambda\mathsf{lxr}$, which can be seen as a functional computational counterpart to Proof Nets. The major features of this calculus are

- Simple syntax and natural semantics via reduction rules and equations;
- Sound and complete correspondence with the Proof Nets model, where the equations and reductions of terms have a natural correspondence with those of Proof Nets;
- Full composition of substitutions;
- Nice properties such as confluence, preservation of strong normalisation, strong normalisation for well-typed terms, and step by step simulation of β-reduction.

Explicit operators and Proof Nets Much work on explicit substitutions has been done in the last 10 years, for example [1,5,8,32]. In particular, an unexpected result was given by Melliès [40] who has shown that there are β-strongly normalisable terms in λ-calculus that are not strongly normalisable when evaluated by the reduction rules of an explicit version of the λ-calculus, such as for example $\lambda\sigma$ [1] or $\lambda\sigma_{\Uparrow}$ [27]. In other words, $\lambda\sigma$ and $\lambda\sigma_{\Uparrow}$ do not enjoy the property known as Preservation of Strong Normalisation (PSN) [5].

This phenomenon shows a defect in the design of these calculi with explicit substitutions because they are supposed to implement their underlying language without losing its good properties. However, there are many ways to avoid Melliès' counter-example in order to recover the PSN property. One of them is to simply forbid the substitution operators to cross lambda-abstractions [39,22]; another consists of avoiding composition of substitutions [5]; another one imposes a simple strategy on the calculus with explicit substitutions to mimic exactly the calculus without explicit substitutions [25]. The first solution leads to *weak* lambda calculi, not able to express *strong* beta-equality, which is used for example in implementations of proof-assistants [12,31]. The second solution is drastic as composition of substitutions is needed in implementations of HO unification [20] or functional abstract machines [28]. The last one does not take benefit of the power of explicit operators because substitutions are neither controlled nor delayed.

In order to cope with this problem David and Guillaume [14] defined a calculus with *explicit labels*, called λ_{ws}, which allows *controlled* composition of substitutions without losing PSN. These labels are obtained by considering a *weakening* rule in the logical system that specifies the typing rules of λ_{ws}, and then by annotating in the term language the formula introduced by this rule via a label. But the λ_{ws}-calculus has a complicated syntax and its named version [17] is even more unreadable. On the positive side we should mention that λ_{ws}-calculus has very nice properties as it is confluent (or Church-Rosser) and enjoys PSN. Also, it can be shown [18] that there is a simple translation from λ_{ws} into the Proof

Nets of Linear Logic that preserves reduction. This translation gives at the same time an elegant proof of strong normalisation for well-typed λ_{ws}-terms. Moreover, the translation reveals a natural semantics for composition of explicit substitutions, and also suggests that explicit erasure and duplication can be added to the calculus without losing termination. These are the main ideas constituting the starting point of the calculus called $\lambda\mathsf{lxr}$ that we present in this paper.

Explicit operators of typed $\lambda\mathsf{lxr}$ have thus a nice logical interpretation: substitution is *cut*, duplication is *contraction*, erasure is *weakening*. From the point of view of implementation, this can be read as the facts that substitution can be delayed, and that duplication and erasure can be controlled.

Instead of translating a term syntax into Proof Nets, we *extract* a term calculus from Proof Nets, thus defining a simple and natural syntax involving not only reduction rules but also *equations*. Every term equation of $\lambda\mathsf{lxr}$ can be seen as a computational counterpart to an equality between Proof Nets and, vice-versa, every Proof Net equality can be naturally read back as an equality between $\lambda\mathsf{lxr}$-terms.

It is then not surprising that we obtain a full correspondence between typed $\lambda\mathsf{lxr}$ and the Intuitionistic fragment of Linear Logic's Proof Nets in the sense that the interpretation is not only *sound* but also *complete* (in contrast to the translation from λ_{ws} to Proof Nets, which was only sound).

Weakening and Garbage Collection The *erasure/weakening operator* has an interesting computational behaviour that we illustrate via an example. Let us denote by $W_{\_}(\_)$ the weakening operator, so that a $\lambda\mathsf{lxr}$-term whose variable x is used to weaken the term N is written $W_x(N)$, that is, we explicitly annotate that the variable x does not appear free in the term N. Then, when evaluating the application of a term $\lambda x.W_x(N)$ to another term L, an explicit substitution $\langle x = L\rangle$ is created and the computation will continue with $W_x(N)\langle x = L\rangle$. Then, the weakening operator will be used to prevent the substitution $\langle x = L\rangle$ from going into the term N, thus making more efficient the propagation of a substitution with respect to the original term.

Another interesting feature of our system is that weakening operators are always *pulled out* to the top-level during $\lambda\mathsf{lxr}$-reduction. Moreover, free variables are *never* lost during computation because they get marked as weakening operators. Indeed, if t β-reduces to t', then its $\lambda\mathsf{lxr}$-interpretation reduces to that of t' where weakening operators are added at the top level to keep track of the variables that are lost during the β-reduction. Thus for example, when simulating the β-reduction steps $(\lambda x.\lambda y.x)Nz \longrightarrow^*_\beta N$, the lost variable z will appear in the result of the computation by means of a weakening operator at the top level, i.e. as $W_z(\overline{N})$ (where $\overline{N}$ is the interpretation of N in $\lambda\mathsf{lxr}$), thus preparing the situation for an efficient garbage collection on z.

The explicit weakening operator can thus be seen as a tool for an efficient implementation of *garbage collection*. This feature is not present in λ-calculus, so one can think that λ-calculus is better or simpler, but implementation of functional programming shows that garbage collection exists and must be taken into account.

It is worth noticing that the labels of the λ_{ws}-calculus cannot be pulled out to the top-level as in $\lambda\mathsf{lxr}$. Also, free variables may be lost during λ_{ws}-computation. Thus, garbage collection within λ_{ws} does not offer the advantages existing in $\lambda\mathsf{lxr}$.

Composition From a rewriting point of view this calculus can be viewed as the first formalism that is confluent (or Church-Rosser) and strongly normalising on typed terms, *simulates β-reduction step by step*, and has PSN as well as *full composition*. By simulation of β-reduction step by step we mean that every β-reduction step in λ-calculus induces a non-empty $\lambda\mathsf{lxr}$-reduction sequence. By full composition we mean that we can compute the application of an explicit substitution to a term, no matter which substitution remains non-evaluated within that term. In particular, in a term $N\langle y = P\rangle\langle x = L\rangle$, the external substitution is not blocked by the internal one and can be further evaluated without ever requiring any preliminary evaluation of $N\langle y = P\rangle$. In other words, the application of the substitution $\langle x = L\rangle$ to the term N can be evaluated independently from that of $\langle y = P\rangle$. A more technical explanation of the concept of full composition appears in Section 2.

Related work Besides the λ_{ws}-calculus [14] and its encoding in linear logic [17] already mentioned, other computational meanings of logic via the use of explicit operators have already been proposed.

Herbelin [30] proposes a term language with applicative terms and explicit substitutions which corresponds to the Gentzen-style sequent calculus LJT. A similar approach to intuitionistic logic is also studied in [49]. In a very different spirit, [10] relates the *pattern matching operator* in functional programming to the cut elimination process in sequent calculus for intuitionistic logic.

Abramsky [2] gives computational interpretations for intuitionistic and classical Linear Logic which are based on sequents rather than Proof Nets. As a consequence, no equalities between terms reflect the fact that some proofs of the sequent calculus approach get identified when expressed as Proof Nets. Many other term calculi based on sequents rather than Proof Nets have been proposed for Linear Logic, as for example [23,6,45,50].

A related approach was independently developed by V. van Oostrom (available in course notes written in Dutch [48]), where explicit operators for contraction and weakening are added to the λ-calculus to present optimal reduction in a framework with implicit substitutions. We show here how the same operators allow a fine control of composition when using explicit substitutions, although the proofs of some fundamental properties, such as PSN and confluence, become harder. A complete overview on optimal sharing in functional programming languages, and its connection with linear logic can be found in [4].

Another approach is taken in [21], where a calculus with explicit operators is defined in order to study the notion of "closed reduction" in λ-calculus. Although reduction rules take enormous advantage of the fact that some subterms are closed (i.e. without free variables), which greatly simplifies the definition of reduction, no deep relation with Proof Nets is exploited and no equalities appear at the level of terms.

Our completeness proof is inspired by [37], where polarised Proof Nets are proposed as a sound and complete model of the $\lambda\mu$ calculus [42]. Finally, a revised version of the calculus λ_{ws} with names is developed in [43].

The paper is organised as follows. Section 2 presents the syntax and operational semantics of the $\lambda\mathsf{lxr}$-calculus. Section 3 defines the model of the calculus and establishes soundness and completeness. Section 4 shows the relation between λ-calculus and $\lambda\mathsf{lxr}$-calculus by giving mutual translations from one to the other. In Section 5 we state the main operational properties of $\lambda\mathsf{lxr}$. Finally we conclude and give some ideas for further work. For lack of space, we cannot give full proofs in this extended abstract; we refer the reader to [33] for further details.

2 The calculus λlxr

The syntax for raw terms, given by the following grammar, is extremely simple [1] and can be just viewed as an extension of that of $\lambda\mathsf{x}$ [8].

$$t ::= x \mid \lambda x.t \mid t\ t \mid t\langle x = t\rangle \mid W_x(t) \mid C_x^{y,z}(t)$$

The term x is called a *variable*, $\lambda x.t$ an *abstraction*, $t\ u$ an *application*, $t\langle x = u\rangle$ a *substitution*, $W_x(t)$ a *weakening* and $C_x^{y,z}(t)$ a *contraction*. The last three constructors are called *explicit operators*.

The terms $\lambda x.t$ and $t\langle x = u\rangle$ define *binders* for the variable x (said to be *bound*) whose scope is t. The term $C_x^{y,z}(t)$ defines a *binder* for y and z (also said to be *bound*) whose scope is t, whereas x is *free* in the terms x, $C_x^{y,z}(t)$ and $W_x(t)$. We write $\mathcal{V}(t)$ to denote the set of variables of the term t and $\mathcal{FV}(t)$ to denote the subset of $\mathcal{V}(t)$ which contains only the free ones. As usual we shall consider α-conversion to guarantee that no variable is free and bound in a term at the same time and that bound variables have all different names.

We say that a term is *linear* if it satisfies the following: in every subterm, every variable has at most one free occurrence, and every binder binds a variable that does have a free occurrence (and hence only one). For instance, the terms $W_x(x)$ and $\lambda x.xx$ are not linear. However, the latter can be represented in the $\lambda\mathsf{lxr}$-calculus by the linear term $\lambda x.C_x^{y,z}(yz)$. More generally, every λ-term can be translated to a linear $\lambda\mathsf{lxr}$-term (c.f. Section 4).

[1] In contrast to λ_{ws} with names [17,18], where terms affected by substitutions have a complex format $t[x, u, \Gamma, \Delta]$

We use Φ, Δ, Σ, $\Pi, \ldots$ to denote finite *lists* of variables (with no repetition). We use the notation $W_{x_1,\ldots,x_n}(t)$ for $W_{x_1}(\ldots W_{x_n}(t))$, and $C^{(y_1,\ldots,y_n),(z_1,\ldots,z_n)}_{x_1,\ldots,x_n}(t)$ for $C^{y_1,z_1}_{x_1}(\ldots C^{y_n,z_n}_{x_n}(t))$. For any term t we define a *renaming operation* $R^{x_1,\ldots,x_n}_{y_1,\ldots,y_n}(t)$ as the result of *simultaneously* substituting y_i for every *free* occurrence x_i in t where $i \in 1 \ldots n$. Thus for instance $R^{x,y}_{x',y'}(C^{y,z}_w(x(yz))) = C^{y,z}_w(x'(yz))$.

We now introduce a congruence $\equiv$ (i.e. a symmetric, reflexive, transitive relation closed under any context) on terms which brings the typed version of the calculus closer to the Proof Nets modulo as defined in [15]. The relation $\equiv$ is the smallest congruence that includes the axioms in Figure 1 expressing associativity and commutativity for contraction (equations $A, C1_c, C2_c$), commutativity for weakening (equation C_w) and commutativity for independent substitutions (equation S), which is also called *parallel composition*. Terms up to rule S could be represented using sets of substitutions instead of atomic ones. Finally, contraction and substitution are treated at the same level using axiom $Cont2$.

$C^{x,v}_w(C^{z,y}_x(t))$	$\equiv_A$	$C^{x,y}_w(C^{z,v}_x(t))$	if $x \neq y, v$
$C^{y,z}_x(t)$	$\equiv_{C1_c}$	$C^{z,y}_x(t)$	
$C^{y',z'}_{x'}(C^{y,z}_x(t))$	$\equiv_{C2_c}$	$C^{y,z}_x(C^{y',z'}_{x'}(t))$	if $x \neq y', z'$ & $x' \neq y, z$
$W_x(W_y(t))$	$\equiv_{C_w}$	$W_y(W_x(t))$	
$t\langle x = u\rangle\langle y = v\rangle$	$\equiv_S$	$t\langle y = v\rangle\langle x = u\rangle$	if $y \notin \mathcal{FV}(u)$ & $x \notin \mathcal{FV}(v)$ & $x \neq y$
$C^{y,z}_w(t)\langle x = u\rangle$	$\equiv_{Cont2}$	$C^{y,z}_w(t\langle x = u\rangle)$	if $x \neq w$ & $y, z \notin \mathcal{FV}(u)$

Figure1. Congruence axioms for λlxr-terms

It can easily be proven that the congruence relation defined by the previous rules preserves free variables and linearity. Since we shall deal with rewriting modulo the congruence $\equiv$, it is worth noticing that $\equiv$ is decidable. More than that, each congruence class contains finitely many terms.

The congruence $\equiv$ enables us to write "$W_{\mathcal{S}}(u)$", or "$C^{\Delta,\Pi}_{\Phi}(t)$ where $\Phi := \mathcal{S}$" without ordering the variables in $\mathcal{S}$. Besides, we shall sometimes not specify what the lists Δ and Π are, assuming them to be two *disjoint* lists of *fresh* variables.

The reduction relation of the calculus, denoted $\longrightarrow_{\lambda\mathsf{lxr}}$, is the relation generated by the reduction rules in Figure 2 modulo the congruence relation in Figure 1. In order to avoid variable capture, rules Abs and $CAbs$ respectively need the side-conditions $(y \notin \mathcal{FV}(u))$ and $(x \neq y, z)$, which can always be satisfied by α-conversion, so that their nature is different from that of the other side-conditions. The rules should be understood in the prospect of applying them to linear terms. Indeed, it can be shown that if t is linear and $t \longrightarrow_{\lambda\mathsf{lxr}} t'$, then t' is linear and $\mathcal{FV}(t) = \mathcal{FV}(t')$. The last statement is achieved in particular by the weakening operator, and coincides with the property called "interface preserving" [36] in interaction nets. The fact that linearity is preserved is a essential requirement of the system, so that we can henceforth consider linear terms only.

It is worth noticing that weakening and contraction can naturally be viewed, respectively, as explicit *erasure* and *duplication* operators. The former may be pulled out to the top level by using rules $WAbs, WApp1, WApp2, WSubs$ in order to eliminate void substitutions as soon as possible, while the latter may be pushed in by using rules $CAbs, CApp1, CApp2, CSubs$ in order to delay the duplication of substitutions as much as possible.

$(\lambda x.t)\ u$	$\longrightarrow_B$	$t\langle x=u\rangle$					
System x				**System r**			
$(\lambda y.t)\langle x=u\rangle$	$\longrightarrow_{Abs}$	$\lambda y.t\langle x=u\rangle$		$\lambda x.W_y(t)$	$\longrightarrow_{WAbs}$	$W_y(\lambda x.t)$	$x \neq y$
$(t\ v)\langle x=u\rangle$	$\longrightarrow_{App1}$	$t\langle x=u\rangle\ v$	$x \in \mathcal{FV}(t)$	$W_y(u)\ v$	$\longrightarrow_{WApp1}$	$W_y(uv)$	
$(t\ v)\langle x=u\rangle$	$\longrightarrow_{App2}$	$t\ v\langle x=u\rangle$	$x \in \mathcal{FV}(v)$	$u\ W_y(v)$	$\longrightarrow_{WApp2}$	$W_y(uv)$	
				$t\langle x=W_y(u)\rangle$	$\longrightarrow_{WSubs}$	$W_y(t\langle x=u\rangle)$	
$x\langle x=u\rangle$	$\longrightarrow_{Var}$	u					
$W_x(t)\langle x=u\rangle$	$\longrightarrow_{Weak1}$	$W_{\mathcal{FV}(u)}(t)$		$C_w^{y,z}(W_y(t))$	$\longrightarrow_{Merge}$	$R_w^z(t)$	
$W_y(t)\langle x=u\rangle$	$\longrightarrow_{Weak2}$	$W_y(t\langle x=u\rangle)$	$x \neq y$	$C_w^{y,z}(W_x(t))$	$\longrightarrow_{Cross}$	$W_x(C_w^{y,z}(t))$	$x \neq y, z$
$C_x^{y,z}(t)\langle x=u\rangle$	$\longrightarrow_{Cont1}$	$C_\Phi^{\Delta,\Pi}(t\langle y=u_1\rangle\langle z=u_2\rangle)$					
			where $\Phi := \mathcal{FV}(u)$	$C_w^{y,z}(\lambda x.t)$	$\longrightarrow_{CAbs}$	$\lambda x.C_w^{y,z}(t)$	
			$u_1 = R_\Delta^\Phi(u)$	$C_w^{y,z}(t\ u)$	$\longrightarrow_{CApp1}$	$C_w^{y,z}(t)\ u$	$y, z \in \mathcal{FV}(t)$
			$u_2 = R_\Pi^\Phi(u)$	$C_w^{y,z}(t\ u)$	$\longrightarrow_{CApp2}$	$t\ C_w^{y,z}(u)$	$y, z \in \mathcal{FV}(u)$
$t\langle y=v\rangle\langle x=u\rangle$	$\longrightarrow_{Comp}$	$t\langle y=v\langle x=u\rangle\rangle$	$x \in \mathcal{FV}(v)$	$C_w^{y,z}(t\langle x=u\rangle)$	$\longrightarrow_{CSubs}$	$t\langle x=C_w^{y,z}(u)\rangle$	$y, z \in \mathcal{FV}(u)$

Figure2. Reduction rules for λlxr-terms

For any reduction relation $\longrightarrow_j$, we denote by $\longrightarrow^+{}_j$ the transitive closure and by $\longrightarrow^*{}_j$ the reflexive and transitive closure.

Owing to the linearity constraints previously imposed, the $Comp$ rule is equivalent to the following rule $Comp_{bad}$:

$$t\langle y=v\rangle\langle x=u\rangle \longrightarrow\ t\langle y=v\langle x=u\rangle\rangle \quad x \notin \mathcal{FV}(t)$$

However, if the linearity constraints are not taken into account, the $Comp_{bad}$ rule causes failure of the PSN property [7].

Also, when linearity constraints are not considered, four cases may occur when composing two substitutions as in $t\langle y=v\rangle\langle x=u\rangle$: either (1) $x \in \mathcal{FV}(t) \cap \mathcal{FV}(v)$, or (2) $x \in \mathcal{FV}(t) \setminus \mathcal{FV}(v)$, or (3) $x \in \mathcal{FV}(v) \setminus \mathcal{FV}(t)$, or (4) $x \notin \mathcal{FV}(t) \cup \mathcal{FV}(v)$.

Composition is said to be *partial* in calculi like λ_{ws} [14] because only cases (1) and (3) are considered by the reduction rules. Because of the linearity constraints of λlxr, cases (1) and (4) have to be dealt with by the introduction of a contraction for case (1) and a weakening for case (4). Those operators will interact with external substitutions by the use of rules ($Weak1$) and ($Cont1$), respectively. Case (3) is treated by rule ($Comp$), and case (2) by the congruence rule $\equiv_S$. We say in this case that composition is *full* as all cases (1)-(4) are treated. Thus, λlxr turns out to be the first term calculus with explicit substitutions having full composition and preserving β-strong normalisation (c.f. Theorem 7).

We now define a typing system for λlxr *Types* are defined by means of the following syntax, where σ ranges over a countable set of atomic types.

$$A ::= \sigma \mid A \rightarrow A$$

An *environment* is a *set* of decorated variables of the form $x : A$, where A is any type. A *judgement* is a triplet of the form $\Gamma \vdash t : A$, where Γ is an environment, t is a λlxr-term, and A is a type.

We say that t is *well-typed* if there is an environment Γ and a type A such that the judgement $\Gamma \vdash t : A$ is derivable from the set of *typing rules* in Figure 3.

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma, x : A, y : A \vdash M : B}{\Gamma, z : A \vdash C_z^{x,y}(M) : B}$$

$$\frac{\Gamma, x : B \vdash t : A \qquad \Delta \vdash M : B}{\Gamma, \Delta \vdash t\langle x = M\rangle : A} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Delta \vdash v : A}{\Gamma, \Delta \vdash (t\ v) : B} \qquad \frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash W_x(t) : A}$$

Figure3. Typing Rules for λlxr-terms

Remark that $\Gamma \vdash t : A$ always implies that the domain of Γ is exactly $\mathcal{FV}(t)$. Also, when writing $\Gamma, x : B$, the variable x is supposed to be fresh w.r.t Γ.

As expected, Subject Reduction holds:

Theorem 1. *If $\Gamma \vdash s : A$ and $s \longrightarrow_{\lambda lxr} s'$, then $\Gamma \vdash s' : A$.*

3 A model for λlxr

This section is devoted to show two of the main properties of our calculus. The first one concerns strong normalisation of well-typed terms, which is achieved by translating well-typed λlxr-terms to MELL Proof Nets. The second one shows that the translation from λlxr to Proof Nets is sound and complete w.r.t the appropriate equivalence relations on terms and proof nets respectively.

We briefly recall here the traditional notion of Proof Nets of Linear Logic and some of its basic properties. We refer the interested reader to [24] for more details.

Let $\mathcal{A}$ be a set of *atomic formulae* equipped with an involutive function $\perp : \mathcal{A} \to \mathcal{A}$, called *linear negation.* The set of formulae of the multiplicative exponential fragment of linear logic (called MELL) is defined as follows:

$$\mathcal{F} \ ::= \ \mathcal{A} \mid \mathcal{F} \otimes \mathcal{F} \mid \mathcal{F} \parr \mathcal{F} \mid !\mathcal{F} \mid ?\mathcal{F}$$

The formula $\mathcal{F} \parr \mathcal{G}$ denotes a "non-economic" version of the classical disjunction, whereas $?\mathcal{F}$ and $!\mathcal{F}$ are used to indicate where contraction or weakening can take place. We extend the notion of *linear negation* to formulae as follows:

$$\begin{array}{ll} (?A)^\perp = !(A^\perp) & (A \otimes B)^\perp = A^\perp \parr B^\perp \\ (!A)^\perp = ?(A^\perp) & (A \parr B)^\perp = A^\perp \otimes B^\perp \end{array}$$

The set of Proof Nets is denoted PN (we refer the reader to [24] or [33] for a formal definition). Proof Nets are the *computational objects* behind Linear Logic, where the notion of reduction (called also "cut elimination") corresponds exactly to the cut-elimination procedure on sequent derivations. The traditional reduction system for MELL consists in *cut elimination rules,* we refer the reader to [24] or [33] for a formal definition.

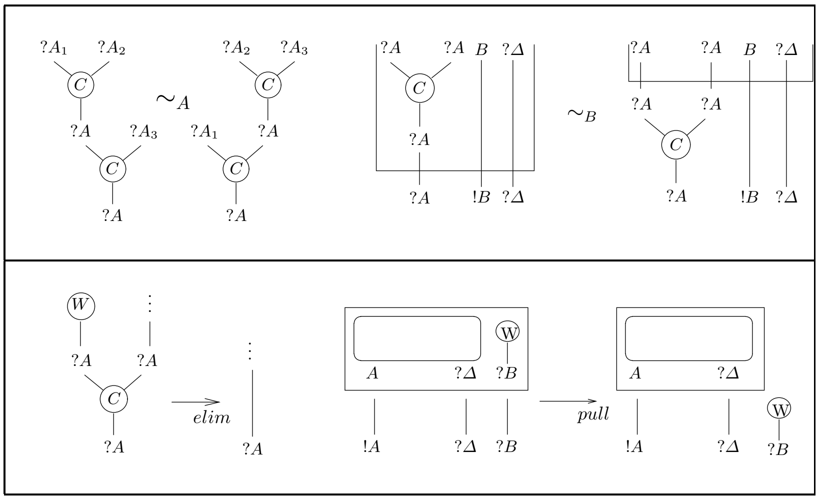

Figure4. Axioms and extra reduction rules for MELL Proof Nets

Unfortunately, the original notion of reduction on PN is not well adapted to simulate either the β rule of λ-calculus, or the rules dealing with propagation of substitution in explicit substitution calculi: too many inessential details about the order of application of the rules are still present, and in order to get rid of them, one is naturally led to define an equivalence relation on PN, as is done in [15], defined by the axioms A and B in Figure 4. Equivalence A turns contraction into an associative operator. Equivalence B abstracts away the relative order of application of the rules of box-formation and contraction. Finally, besides these equivalence relations defined in [15], we shall also need the two extra reduction rules in Figure 4: *elim* is used to remove unneeded weakening links when simulating explicit substitutions and *pull* allows weakening links to go outside boxes.

Notation: Henceforth, we shall call R the system made of rules *Ax-cut*, $⅋$-$\otimes$, *w-b*, *d-b*, *c-b*, *b-b* and *elim* and *pull*. We shall write $\sim_E$ for the congruence (reflexive, symmetric, transitive, closed by contexts) relation on Proof Nets generated by axioms A, B. We shall write R/E for the system made of the rules in R and the axioms in A, B. System R/E is actually defining a notion of *reduction modulo* *an equivalence relation*, so we write $r \longrightarrow_{R/E} s$ if and only if there exist r' and s' such that $r \sim_E r' \longrightarrow_R s' \sim_E s$.

In order to prove the properties of the translation from typed λlxr to Proof Nets, we shall use the following result:

Theorem 2. *The reduction relation $\longrightarrow_{R/E}$ terminates.*

Proof. This result is proved in [43] for which we refer the interested reader for full details. The proof uses postponement of rule *pull* w.r.t to the rest of the system for which termination is stated in [15]. For the sake of completeness, we include the proof in English in the full version of our paper [33].

We now present the natural interpretation of typed λlxr-terms as Proof Nets. For that, we use the translation of types introduced in [13] given by :

$$\begin{array}{lll} A^* & = A & \text{for atomic types} \\ (A \to B)^* & = ?((A^*)^\perp) \parr B^* & \text{otherwise} \end{array}$$

Figure 5 defines the translation $T(\_)$ from derivable typing judgements of λlxr to Proof Nets. We shall often write $T(t)$ instead of $T(\Gamma \vdash t : A)$ when Γ and A do not matter or are clear (from Subject Reduction, for example). $T(\_)$ satisfies the following properties:

Lemma 1 (Simulation of λlxr-reduction).

- *If $t \equiv t'$, then $T(t) \sim_E T(t')$.*
- *If $t \longrightarrow_B t'$, then $T(t) \longrightarrow^+{}_{R/E} T(t')$.*
- *If $t \longrightarrow_{\mathsf{xr}} t'$, then $T(t) \longrightarrow^*{}_{R/E} T(t')$.*

As a consequence we obtain one of the main important properties of λlxr:

Theorem 3 (Strong Normalisation).
The relation $\longrightarrow_{\lambda\mathsf{lxr}}$ is strongly normalising on well-typed λlxr-terms.

Proof. An infinite λlxr-reduction sequence would have infinitely many B-steps. Indeed, system xr can be proven terminating [33]. But this would lead by Lemma 1 to an infinite R/E-reduction sequence which is impossible by Theorem 2.

In order to show Theorem 3 another technique [43] using Preservation of Strong Normalisation (c.f. Section 5) together with the strong normalisation of typed λ-calculus could be used. Direct proofs using for instance reducibility by perpetuality [9,38] seem much more difficult to adapt to our case. We remark however that those techniques are no simpler than ours, as many auxiliary properties about the computational behaviour of the calculus need to be establish.

The relevant computational steps of the cut-elimination procedure in Proof Nets are those modifying the box structure. Hence, we are interested in characterising those terms that are translated by $T(\_)$ into Proof Nets that have identical box structures. Let TB be the reduction relation on PN generated by the rules that do not modify the box structure, namely *pull*, *elim*, *Ax-cut* and $\parr$-$\otimes$, modulo the congruence $\sim_E$.

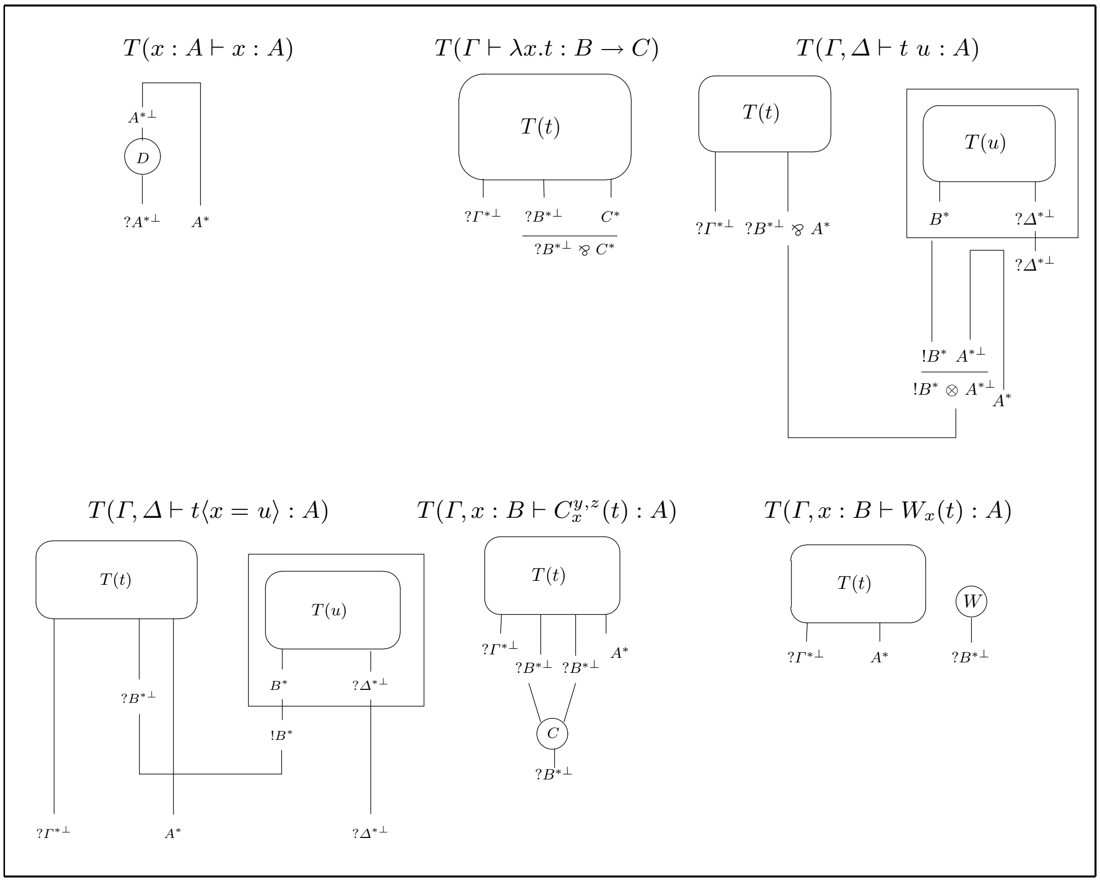

Figure5. Encoding typed λlxr-terms into MELL proof-nets

Proposition 1. *The reduction relation TB is confluent and terminating. Hence, the normal form of a proof net r w.r.t this reduction relation, written $TB(r)$, exists and is unique up to the congruence $\sim_E$.*

Hence, "having the same box structure" can be expressed by the equivalence relation $\approx$ defined as: $r \approx r'$ if and only if $TB(r) \sim_E TB(r')$.

We now define for the terms a congruence $\cong$ obtained by adding to $\equiv$ the following rules turned into equalities:

$$\{B, Abs, App1, Weak2, WAbs, WApp1, WApp2, Cross, Merge, CAbs, CApp1, CApp2\}$$

Remark that $WSubs$ and $CSubs$ are included in $\cong$.

The following property relates two $\cong$-convertible terms w.r.t. their semantic translations into Proof Nets and is one of the interesting results about λlxr.

Theorem 4 (Soundness and Completeness). *The interpretation $T(\_)$ is sound and complete, i.e. given two λlxr-terms t_1, t_2 we have $t_1 \cong t_2$ iff $T(t_1) \approx T(t_2)$.*

The proof is inspired by that of a similar characterisation, given in [37], for $\lambda\mu$-terms with respect to Polarized Proof Nets, where equality in the term syntax is an extension of the σ-equivalence on λ-terms defined in [44]. Yet, the latter needs to consider specific permutations of β-redexs to achieve the characterisation, instead of simply turning some reduction rules into equivalence rules.

4 Recovering the λ-calculus

We show in this section the relation between $\lambda\mathsf{lxr}$-terms and λ-terms. More precisely, we show that the linearity constraints and the use of explicit resource operators in $\lambda\mathsf{lxr}$ are sufficient to decompose the β-reduction step into smaller steps. We shall use the notation $\Gamma \vdash_\lambda t : A$ to denote typing judgements and typing derivability in λ-calculus.

We now describe how to encode a λ-term (possibly not linear) into a $\lambda\mathsf{lxr}$ one, up to the congruence $\equiv$.

Definition 1. *The encoding of λ-terms is defined by induction as follows:*

$$\begin{array}{lll}
\mathcal{A}(x) & := x & \\
\mathcal{A}(\lambda x.t) & := \lambda x.\mathcal{A}(t) & \textit{if } x \in \mathcal{FV}(t) \\
\mathcal{A}(\lambda x.t) & := \lambda x.W_x(\mathcal{A}(t)) & \textit{if } x \notin \mathcal{FV}(t) \\
\mathcal{A}(tu) & := C^{\Delta,\Pi}_{\Phi}(R^{\Phi}_{\Delta}(\mathcal{A}(t))\ R^{\Phi}_{\Pi}(\mathcal{A}(u))) & \textit{where } \Phi := \mathcal{FV}(t) \cap \mathcal{FV}(u)
\end{array}$$

Note that $\mathcal{A}(tu) = \mathcal{A}(t)\mathcal{A}(u)$ in the particular case $\mathcal{FV}(t) \cap \mathcal{FV}(u) = \emptyset$.

It is worth noticing that $\mathcal{A}$ commutes with renaming (i.e. $\mathcal{A}(R^{\Phi}_{\Delta}(t)) = R^{\Phi}_{\Delta}(\mathcal{A}(t))$) and that $\mathcal{A}$ preserves free variables (i.e. $\mathcal{FV}(t) = \mathcal{FV}(\mathcal{A}(t))$). As a consequence, the encoding of a λ-term is a linear $\lambda\mathsf{lxr}$-term. For instance, given $t = \lambda x.\lambda y.y(zz)$, we have $\mathcal{A}(t) = \lambda x.W_x(\lambda y.(y\ C^{z_1,z_2}_{z}(z_1\ z_2)))$.

Notice that a β-reduction step may not preserve the set of free variables whereas any reduction in $\lambda\mathsf{lxr}$ does. Indeed, we have $t = (\lambda x.y)\ z \longrightarrow_\beta\ y$, but

$$\mathcal{A}(t) = (\lambda x.W_x(y))\ z \longrightarrow^*_{\lambda\mathsf{lxr}} W_z(y) = W_z(\mathcal{A}(y))$$

It is hence the following statement that we prove by induction on terms.

Theorem 5. *If $t \longrightarrow_\beta\ t'$, then $\mathcal{A}(t) \longrightarrow^+_{\lambda\mathsf{lxr}} W_{\mathcal{FV}(t)\setminus\mathcal{FV}(t')}(\mathcal{A}(t'))$.*

As for the types, a straightforward induction on typing derivations allows us to show the soundness of the translation $\mathcal{A}$:

Lemma 2 ($\mathcal{A}$ preserves types). *If $\Gamma \vdash_\lambda t : A$, then $\Gamma \vdash W_{\Gamma\setminus\mathcal{FV}(t)}(\mathcal{A}(t)) : A$.*

We now show how to encode a $\lambda\mathsf{lxr}$-term into a λ-term.

Definition 2. *Let t be a λlxr-term. We define the function $\mathcal{B}(t)$ by induction on the structure of t as follows:*

$$\begin{array}{llll} \mathcal{B}(x) & = x & \mathcal{B}(t\langle x = u\rangle) & = \mathcal{B}(t)\{x\backslash\mathcal{B}(u)\} \\ \mathcal{B}(\lambda x.t) & = \lambda x.\mathcal{B}(t) & \mathcal{B}(t\ u) & = \mathcal{B}(t)\mathcal{B}(u) \\ \mathcal{B}(W_x(t)) & = \mathcal{B}(t) & \mathcal{B}(C_x^{y,z}(t)) & = \mathcal{B}(t)\{y\backslash x\}\{z\backslash x\} \end{array}$$

The translation $\mathcal{B}$ commutes with renaming (i.e. $\mathcal{B}(R_\Delta^\Phi(t)) = R_\Delta^\Phi(\mathcal{B}(t))$) and does not add new free variables (i.e. $\mathcal{FV}(\mathcal{B}(t)) \subseteq \mathcal{FV}(t)$). Now the following simulation result can be proven:

Theorem 6. *If $M \longrightarrow_{\lambda\mathsf{lxr}} N$, then $\mathcal{B}(M) \longrightarrow^*_\beta \mathcal{B}(N)$.*

Remark that congruent terms are mapped to the same λ-term, so that it makes sense to consider $\mathcal{B}(\mathcal{A}(\_))$, which is in fact the identity: $t =_\alpha \mathcal{B}(\mathcal{A}(t))$.

A straightforward induction on typing derivations allows us to show:

Lemma 3 ($\mathcal{B}$ preserves types). *If $\Gamma \vdash t : A$, then $\Gamma \vdash_\lambda \mathcal{B}(t) : A$.*

5 Operational Properties

In Sections 2, 3 and 4 we have already established the properties of subject reduction, strong normalisation of well-typed λlxr-terms and simulation of β-reduction step by step. But a calculus which is defined in order to implement λ-calculus is also expected to preserve fundamental properties such as confluence and preservation of strong normalisation. We state in this section both properties which hold not only for well-typed terms but for all (linear) terms.

The original notion of PSN [5] has to be properly reformulated in our context as follows: every strongly normalisable λ-term *is encoded* into a strongly normalisable λlxr-term. We establish PSN of λlxr by simulating reductions in λlxr by reductions in the λI-calculus of Church-Klop [35] with its associated reduction relations β,π. We refer the reader to [47,51] for a survey on different techniques based on the λI-calculus to infer normalisation properties. Our proof technique can be summarised as follows (full details are given in [33]).

1. Define a relation $\mathcal{I}$ between linear λlxr-terms and λI-terms.
2. Show that $t\,\mathcal{I}\,T$ and $t \longrightarrow_{\mathsf{xr}} t'$ imply $t'\,\mathcal{I}\,T$,
3. Show that $t\,\mathcal{I}\,T$ and $t \longrightarrow_B t'$ imply $\exists T'$ such that $t'\,\mathcal{I}\,T'$ and $T \longrightarrow^+_{\beta\pi} T'$.
4. Deduce from 1,2,3 that if $t\,\mathcal{I}\,T$ and $T \in SN_{\beta\pi}$, then $t \in SN_{\lambda\mathsf{lxr}}$.
5. Define an encoding $i() : \lambda \mapsto \lambda I$ such that if $t \in SN_\beta$ then $i(t) \in WN_{\beta\pi}$.
6. Show that $\mathcal{A}(t)\,\mathcal{I}\,i(t)$, where $\mathcal{A}(t)$ is the encoding given in Section 4.

Theorem 7 (PSN). *For any λ-term t, if $t \in SN_\beta$, then $\mathcal{A}(t) \in SN_{\lambda\mathsf{lxr}}$.*

Proof. If $t \in SN_\beta$, then by the above point 5, $i(t) \in WN_{\beta\pi}$. A well-known result of Nederpelt [41] states that $WN_{\beta\pi} \subseteq SN_{\beta\pi}$, so $i(t) \in SN_{\beta\pi}$ and by points 6 and 4 we have $\mathcal{A}(t) \in SN_{\lambda\mathsf{lxr}}$.

We now use both simulations presented in Section 4 to derive the confluence property via a generalisation of the Interpretation Method [26]. We refer the reader to [33] for more details.

Theorem 8. *The system* $\lambda\mathsf{lxr}$ *is confluent.*

6 Conclusion and further work

This paper extends the explicit substitution paradigm by showing how the Proof Nets of Linear Logic can be suitable as a logical model of a language with explicit operators for erasure, duplication and substitution.

Our term language is expressed by a simple syntax, and enjoys natural operational semantics via a well-established notion of reduction modulo a set of equations. Soundness and completeness of $\lambda\mathsf{lxr}$ are shown with respect to its Proof Nets model.

In contrast to other calculi in the literature, $\lambda\mathsf{lxr}$ has full composition and enjoys PSN. Moreover, $\lambda\mathsf{lxr}$ enjoys all the nice properties that one expects such as confluence, strong normalisation of well-typed terms and step by step simulation of β-reduction. All these properties are shown by considering the complex notion of reduction modulo an equivalence which we have associated to $\lambda\mathsf{lxr}$-terms.

We claim that weakening operators are well-adapted to implement efficient garbage collection. Indeed, free variables are never lost and weakening operators are pulled out to the top-level during computation.

Our soundness and completeness proofs illustrate that the only rules with computational relevance in $\lambda\mathsf{lxr}$ are $\{App2, Comp, Var, Weak1, Cont1\}$, just as the interesting rules in Proof Nets are only those concerning boxes. More precisely, $App2$ and $Comp$ in $\lambda\mathsf{lxr}$ correspond to *b*-*b* in PN, Var to *d*-*b*, $Weak1$ to *w*-*b* and $Cont1$ to *c*-*b*.

It is worth mentioning the calculus obtained by turning the equation $Cont2$ into a reduction rule (from left to right) and by eliminating reduction rules $WSubs$ and $CSubs$ enjoys exactly the same properties as the calculus presented in this paper, namely Theorems 1,3,5,6,7,8. However, they seem to be necessary for the confluence on open terms (ongoing work).

We think that many interesting points raised in this work deserve further development. The first one concerns the study of reduction strategies well-adapted to handle explicit operators of substitution, erasure and duplication. This may take into account the notion of weak reduction used to implement functional programming [39].

Proof techniques used in the literature to show PSN of calculi with explicit substitutions (zoom-in [3], minimality [5], semantic RPO [7], PSN by standardisation [34], or intersection types [19]) are not all easy to adapt/extend to $\lambda\mathsf{lxr}$ and other formalisms. We believe that the proof technique used here is really flexible.

But using the PSN result, we believe that we can characterise very neatly the strongly normalising terms of λlxr as the terms typable with intersection types, as it it the case in λ-calculus as well as in the explicit substitution calculus $\lambda\mathtt{x}$ [38].

First-order term syntax for λlxr via de Bruijn indices, or other special notation to avoid α-conversion as for example explicit scoping [29] or also director strings [46], would make implementation easier and bring the term language even closer to the Proof Nets model which has no notion of binding.

Acknowledgements

We are very grateful to R. Dyckhoff, J. Forest, B. Guillaume, O. Laurent, P. Lescanne, J. Mc Kinna and V. van Oostrom for valuable comments and suggestions.

References

1. M. Abadi, L. Cardelli, P. L. Curien, and J.-J. Lévy. Explicit substitutions. *JFP*, 4(1):375–416, 1991.
2. S. Abramsky. Computational interpretations of linear logic. *TCS*, 111:3–57, 1993.
3. A. Arbiser, E. Bonelli, and A. Ríos. Perpetuality in a lambda calculus with explicit substitutions and composition. *WAIT*, *JAIIO*, 2000.
4. A. Asperti and S Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
5. Z.-E.-A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. $\lambda\upsilon$, a calculus of explicit substitutions which preserves strong normalisation. *JFP*, 6(5):699–722, 1996.
6. N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. *TLCA*, *LNCS* 664, pages 75–90, 1993.
7. R. Bloo and H. Geuvers. Explicit substitution: on the edge of strong normalization. *TCS*, 211(1-2):375–395, 1999.
8. R. Bloo and K. Rose. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. In *Computing Science in the Netherlands*, pages 62–72, Netherlands Computer Science Research Foundation, 1995.
9. E. Bonelli. Perpetuality in a named lambda calculus with explicit substitutions, *MSCS*, 11(1):409–450, 2001.
10. S. Cerrito and D. Kesner. Pattern matching as cut elimination. *LICS*, pages 98–108, 1999.
11. A. Church. The calculi of lambda conversion, Princeton University Press, 1941.
12. The Coq Proof Assistant. `http://coq.inria.fr/`.
13. V. Danos, J.-B. Joinet, and H. Schellinx. Sequent calculi for second order logic. *Advances in Linear Logic*, Cambridge University Press, 1995.
14. R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *MSCS*, 11:169–206, 2001.
15. R. Di Cosmo and S. Guerrini. Strong normalization of proof nets modulo structural congruences. *RTA*, *LNCS* 1631, pages 75–89, 1999.
16. R. Di Cosmo and D. Kesner. Strong normalization of explicit substitutions via cut elimination in proof nets. *LICS*, pages 35–46, 1997.
17. R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. *FOSSACS*, *LNCS* 1784, pages 63–81, 2000.
18. R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. *MSCS*, 13(3):409–450, 2003.
19. D. Dougherty and P. Lescanne. Reductions, Intersection Types and Explicit Substitutions. *TLCA*, *LNCS* 2044, pages 121-135, 2001.

20. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *LICS*, 1995.
21. M. Fernández and I. Mackie. Closed reductions in the lambda calculus. *CSL*, *LNCS* 1683, 1999.
22. J. Forest. A weak calculus with explicit operators for pattern matching and substitution. *RTA*, *LNCS* 2378, pages 174–191, 2002.
23. N. Ghani, V. de Paiva, and E. Ritter. Linear explicit substitutions. *IGPL*, 8(1):7–31, 2000.
24. J.-Y. Girard. Linear logic. *TCS*, 50(1):1–101, 1987.
25. J. Goubault-Larrecq. A proof of weak termination of typed lambda sigma-calculi. *TYPES*, *LNCS* 1512, pages 134–151, 1996.
26. T. Hardin. Résultats de confluence pour les règles fortes de la logique combinatoire catégorique et liens avec les lambda-calculs. PhD Thesis, Université Paris 7, 1987.
27. T. Hardin and J.-J. Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989.
28. T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the lambda-sigma calculus. *ICFP*, 1996.
29. D. Hendriks and V. van Oostrom. Adbmal. *CADE*, *LNAI* 2741, pages 136–150, 2003.
30. H. Herbelin. A λ-calculus structure isomorphic to sequent calculus structure. *CSL*, *LNCS 933*, 1994.
31. The HOL system. `http://www.dcs.gla.ac.uk/~tfm/fmt/hol.html`.
32. F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. *PLILP*, *LNCS* 982, pages 45–62, 1995.
33. D. Kesner and S. Lengrand. An Explicit Operator Calculus as the Syntactic Counterpart to a Proof-Net Model. Available at `http://www.pps.jussieu.fr/~kesner/papers`, 2004.
34. Z. Khasidashvili, M. Ogawa, V. van Oostrom. Uniform Normalization Beyond Orthogonality. *RTA*, *LNCS* 2051, pages 122–136, 2001.
35. J.-W. Klop. *Combinatory Reduction Systems*, PhD Thesis, volume 127 of *Mathematical Centre Tracts*. CWI, Amsterdam, 1980.
36. Y. Lafont. Interaction Nets. *POPL*, pages 95–108, 1990.
37. O. Laurent. Polarized proof-nets and lambda-mu calculus. *TCS*, 1(290):161–188, 2003.
38. S. Lengrand. P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini and S. van Bakel. Intersection types for explicit substitutions. *I & C*, 189(1):17–42, 2004.
39. J.-J. Lévy and L. Maranget. Explicit substitutions and programming languages. *FSTTCS*, *LNCS* 1738, pages 181–200, 1999.
40. P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. *TLCA*, *LNCS* 902, pages 328–334, 1995.
41. R. Nederpelt. Strong Normalization in a Typed Lambda Calculus with Lambda Structured Types. PhD Thesis, *Eindhoven University of Technology*, 1973.
42. M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. *LPAR*, *LNCS* 624, pages 190–201, 1992.
43. E. Polonovski. Substitutions explicites et preuves de normalisation. PhD thesis, Université Paris 7, 2004.
44. L. Regnier. Une équivalence sur les lambda-termes. *TCS*, 2(126):281–292, 1994.
45. S. Ronchi della Rocca and L. Roversi. Lambda calculus and intuitionistic linear logic. *Studia Logica*, 59(3), 1997.
46. F.-R. Sinot and M. Fernández and I. Mackie. Efficient Reductions with Director Strings. *RTA*, *LNCS* 2706, pages 46–60, 2003.
47. M.H. Sorensen. Strong Normalization From Weak Normalization in Typed Lambda-Calculi. *I&C*, 37:35–71, 1997.
48. V. van Oostrom. Net-calculus. Course Notes available on `http://www.phil.uu.nl/~oostrom/typcomp/00-01/net.ps`, 2001.
49. R. Vestergaard and J. Wells. Cut Rules and Explicit Substitutions. *MSCS*, 11(1), 2001.
50. P. Wadler. A syntax for linear logic. *MFPS*, *LNCS* 802, pages 513–529, 1993.
51. H. Xi. Weak and Strong Beta Normalisations in Typed Lambda-Calculi. *TLCA*, *LNCS* 1210, pages 390–404, 1997.