Explicit Substitutions for Linear Logical Frameworks: *Preliminary Results* *

Iliano Cervesato
Computer Science Department
Stanford University
Stanford, CA 94305–9045 — USA
iliano@cs.stanford.edu

Valeria de Paiva and Eike Ritter
School of Computer Science
University of Birmingham
Birmingham, B15 2TT — UK
{*V.DePaiva* | *E.Ritter*}*@cs.bham.ac.uk*

Abstract

We present the calculus xdLLF$^-$ and experiment with aspects of its meta-theory. xdLLF$^-$ integrates linear explicit substitutions in de Bruijn notation into the simply-typed fragment of the linear logical framework LLF. After observing that the expected σ-rules invalidate subject reduction, we devise a specification of σ-normalization inspired by the big-step semantics of programming languages, and prove it correct.

1 Introduction

Explicit substitutions [1] have been used to rationalize the implementation of many systems based on various λ-calculi, such as functional languages, logical frameworks, and higher-order logic programming languages. As linear λ-calculi have grown in popularity, so has the need for solid and efficient support for their implementation. A linear adaptation of explicit substitution techniques is a prime candidate. The authors of this paper have separately explored this possibility in two distinct settings:

- In [6], Ghani, de Paiva, and Ritter have designed the language xDILL, geared towards the implementation of functional languages. It is based on Barber and Plotkin's DILL (Dual Intuitionistic Linear Logic) [2], and is characterized, among other things, by variables of two different kinds: linear variables are used exactly once, and intuitionistic variables can be accessed arbitrarily many times. The extra information about usage of linear variables makes it possible to apply various optimizations like update-in-place of aggregate data structures such as arrays, or savings in memory allocation. This significantly influenced the design decisions of the calculus in [6].
- On the other hand, Cervesato and Pfenning have based their implementation of the linear logical framework LLF [4] on a form of linear explicit substitution, although they did not thoroughly investigate its meta-theory. LLF is a close relative of DILL (for example, both distinguish linear and intuitionistic variables). LLF is however designed as a logical framework, which forces a set of operations on terms that are not found in DILL. An implementation of LLF must support

*The first author was partially supported by DoD MURI, "Semantic Consistency in Information Exchange" as ONR Grant N00014-97-1-0505. The second and third authors were partially funded by ESPSRC grant GR/28296, "The eXplicit Substitution Linear Abstract Machine".

term reconstruction to make meta-representation practical, permit logic programming-style proof-search, and accommodate the forthcoming addition of theorem proving capabilities. Each of these functionalities relies on (higher-order) unification and therefore an explicit substitution calculus for LLF must handle meta-variables and their manipulation.

In this paper, we bring our experiences together in trying to isolate some of the issues that arise when combining linearity and explicit substitutions in our different settings. Although the results reported here are very preliminary, this work had the effect of furthering our understanding of these problems.

We start from the linear λ-calculus LLF$^-$ (Section 2), which includes operators from both LLF and DILL while ignoring complex features such as dependent types and an unrestricted "!" operator. LLF$^-$ enjoys properties such as subject reduction, normalization and confluence. Then, by using a standard process [1], we construct the calculus xdLLF$^-$ (Section 3) which incorporates substitutions as a separate syntactic category in LLF$^-$ (along the way, we also switch to a de Bruijn notation, motivated by our interests in efficient implementations). This has the positive effect of turning the meta-level substitutions produced by the β-reductions into explicit substitutions that can be manipulated within the calculus. At this point, the standard approach [1] would require us to express the implicit procedure to carry out the application of a meta-level substitution as a set of rewrite rules about explicit substitutions (σ-rules): the transcription is correct if we can prove that there is a reduction strategy which eliminates all explicit substitutions and terminates with the λ-term that would be produced by making all explicit substitutions implicit.

We deviate from this path since the σ-rules we would obtain for xdLLF$^-$ interfere with linearity and allow rewriting well-typed terms into ill-typed objects. In [6], we solved this problem by splitting the linear substitutions according to the usage of the linear variables. This approach may cause a significant overhead when implemented, and does not scale up when extending xdLLF$^-$ with metavariables. We instead explore a different path (Section 4): we give a syntactic characterization of the set of σ-normal terms (to which no σ-rule would be applicable) as the language xdLLF$^-_\sigma$, outline a type-preserving procedure that reduces a typable xdLLF$^-$ term to its σ-normal form, and prove its correctness. Although this approach deals correctly with linearity, it still has several drawbacks. First, it fixes the reduction strategy, which is instead open when σ-rules are used. Second, it does not allow interleaving σ-normalization steps with other reductions. Third, it does not scale up to handle meta-variables. Nonetheless, we see it as a valuable first step toward addressing these issues more satisfactorily (Section 5).

2 LLF$^-$

The calculus LLF$^-$, that we use as our starting point, enriches the simply-typed fragment of the language of the linear logical framework LLF [4] with multiplicative pairs and unit. On the other hand, it extends the language DILL [2, 6] with additives and with intuitionistic functions, but sacrifices its full-fledged exponential "!". LLF$^-$ is defined as follows:

Types:			*Terms:*					
$A ::=$	a		$M ::=$	x				
	$\top$			$\langle\rangle$				(*additive unit*)
	$A_1 \,\&\, A_2$			$\langle M_1, M_2\rangle$	$\mid$	$\mathsf{fst}\, M$	$\mid\ \mathsf{snd}\, M$	(*additive pairs*)
	$\mathbf{1}$			$\bullet$	$\mid$	$\mathsf{let}\, M_1\, \mathsf{be}\, \bullet\, \mathsf{in}\, M_2$		(*multiplicative unit*)
	$A_1 \otimes A_2$			$M_1 \otimes M_2$	$\mid$	$\mathsf{let}\, M_1\, \mathsf{be}\, x_1 \otimes x_2\, \mathsf{in}\, M_2$		(*multiplicative pairs*)
	$A_1 \multimap A_2$			$\hat{\lambda} x \hat{:} A.\, M$	$\mid$	$M_1 \hat{} M_2$		(*linear functions*)
	$A_1 \to A_2$			$\lambda x : A.\, M$	$\mid$	$M_1\, M_2$		(*intuitionistic functions*)

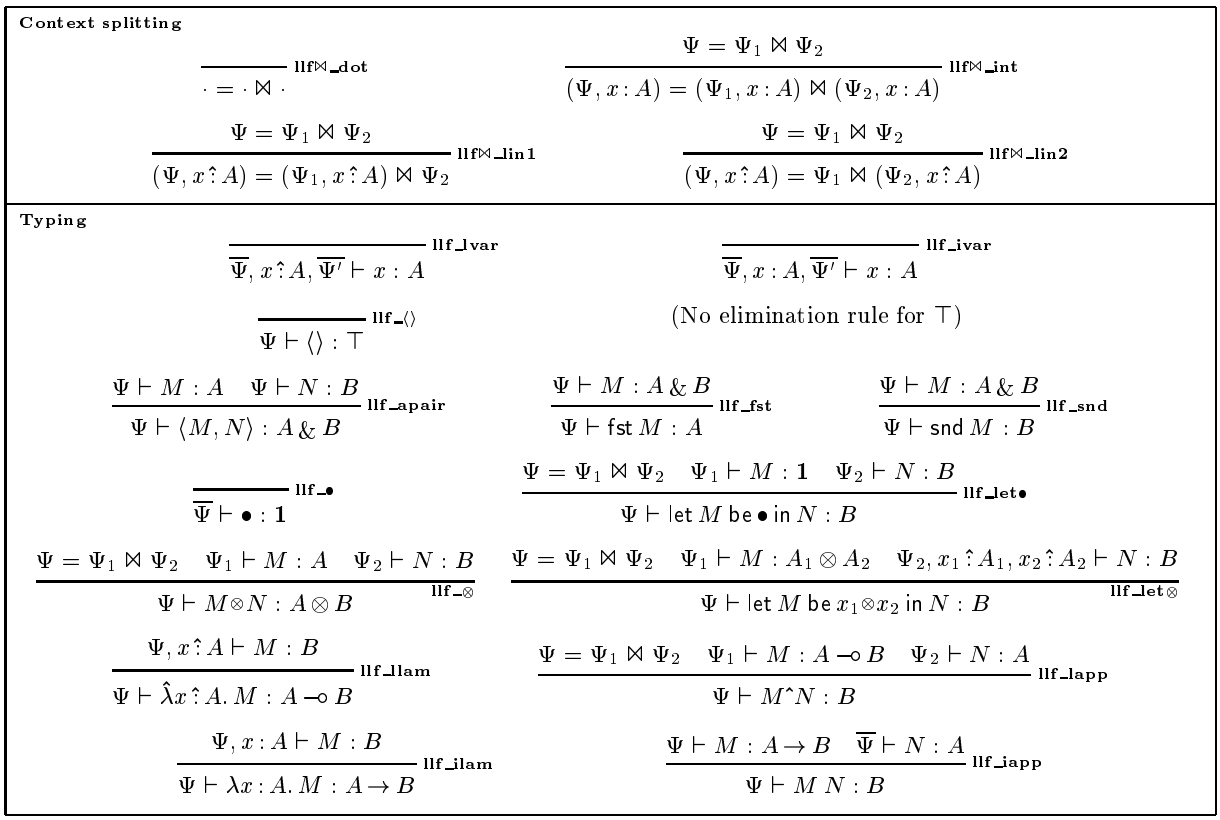

Figure 1: Typing in LLF$^-$

Here x and a range over variables and base types, respectively. In addition to the names displayed above, we will often use N and B for terms and types, respectively. As usual, we rely on a context to assign types to free variables.

$$\textit{Contexts:} \quad \Psi ::= \cdot \mid \Psi, x:A \mid \Psi, x\hat{:}A$$

where $x\hat{:}A$ and $x:A$ stand for a linear and a reusable (intuitionistic) assumption of type A, respectively.

The notions of free and bound variables are adapted from the simply typed λ-calculus. As usual, we identify terms that differ only by the name of their bound variables. Contexts are treated as sequences, we promote "," to denote their concatenation and omit writing "$\cdot$" when unnecessary. As usual, we require variables to be declared at most once in a context. Finally, we write $\overline{\Psi}$ for the intuitionistic part of context Ψ. It is obtain by removing every linear declaration $x\hat{:}A$ from Ψ. See [4] for a formal definition.

The typing judgment for LLF$^-$ has the form $\Psi \vdash M : A$ (read "*M has type A in Ψ*") and is defined in Figure 1. It relies on the auxiliary context splitting judgment $\Psi = \Psi_1 \bowtie \Psi_2$. Due to space reasons, we shall refer the reader to [4] for a discussion of these rules.

The rewriting semantics of LLF$^-$ is given by the usual β-reductions, commuting conversions (generated by the two forms of let), and, depending on one's taste, η-rules. We will only marginally be concerned with these various rules in the sequel. Definitions and properties of interest can be extrapolated from [4]

and [6], or found in [3].

A nameless variant of LLF^- is obtained by straightforwardly extending the standard de Bruijn transformation [5]. As in the case of the λ-calculus, this translation preserves typing and reductions. The resulting calculus, dLLF^-, is at the basis of our current experimentation with explicit substitutions. Space reasons prevent us from discussing it further (see [3] for more). However, its language and typing rules correspond exactly to the term fragment of the σ-normal calculus xdLLF^-_σ discussed in Section 4.

3 xdLLF$^-$

In [6], we devised a calculus of linear explicit substitutions based on named variables. Here, we instead investigate a variant in the style of [1] that uses the de Bruijn notation (this is mainly motivated by implementation considerations). Even in this restricted setting, there are many ways to incorporate explicit substitution into LLF^- (or dLLF^-). In designing xdLLF^-, we chose to model (normal) substitutions on the structure of contexts. We mention other possibilities in Section 5.

The types of xdLLF^- are the same as LLF^-. Its term constructors are adapted from this language as done in [1]. Substitutions may contain the linear extension operator "$\hat{\cdot}$" to account for terms to be substituted for linear variables. Since de Bruijn numbers are positional indices in a substitution (and a context), we use "$\_$" to mark a term that has already been linearly substituted. Terms and substitutions are defined by the following grammar:

Terms:

$t ::=$	1			(*variable indices*)
$\mid$	$\langle\rangle$			(*additive unit*)
$\mid$	$\langle t_1, t_2\rangle$	$\mid$	$\mathsf{fst}\, t \mid \mathsf{snd}\, t$	(*additive pairs*)
$\mid$	$\bullet$	$\mid$	$\mathsf{let}\bullet\, t_1\, \mathsf{in}\, t_2$	(*multiplicative unit*)
$\mid$	$t_1 \otimes t_2$	$\mid$	$\mathsf{let}\otimes\, t_1\, \mathsf{in}\, t_2$	(*multiplicative pairs*)
$\mid$	$\hat{\lambda}_A.\, t$	$\mid$	$t_1 \hat{} t_2$	(*linear functions*)
$\mid$	$\lambda_A.\, t$	$\mid$	$t_1\, t_2$	(*intuitionistic functions*)
$\mid$	$t[\sigma]$			(*substitution application*)

Substitutions:

$\sigma ::=$	Id		(*identity*)
$\mid$	$\uparrow$		(*shift*)
$\mid$	$t \hat{} \sigma$		(*linear extension*)
$\mid$	$\_ \hat{} \sigma$		(*used linear extension*)
$\mid$	$t\,.\,\sigma$		(*intuitionistic extension*)
$\mid$	$\sigma_1 \circ \sigma_2$		(*composition*)

In addition to t and σ, we will use s and τ to denote xdLLF^- terms and substitutions, respectively. Contexts in xdLLF^- are the nameless variant of LLF^- contexts, with again the marker "$\_$" to account for the positional nature of de Bruijn indices when dealing with used assumptions.

$$\textit{Contexts:} \qquad \Gamma \;::=\; \cdot \;\mid\; \Gamma \hat{}, A \;\mid\; \Gamma \hat{}, \_ \;\mid\; \Gamma, A$$

As in LLF^-, we write $\overline{\Gamma}$ to indicate the intuitionistic portion of Γ. It is obtain by replacing every linear assumption with "$\_$".

The typing judgments for terms and substitutions are denoted $\Gamma \vdash_{\mathsf{xd}} t : A$ (read "*t has type A in Γ*") and $\Gamma \vdash_{\mathsf{xd}} \sigma : \Gamma'$ (read "*σ maps terms from Γ' to Γ*"), respectively. As for LLF^-, their definition relies on the auxiliary context splitting judgment $\Gamma = \Gamma_1 \bowtie \Gamma_2$. The rules for these three judgments are displayed in Figure 2.

Rewrite rules in the β, commuting and possibly η families are adapted from LLF^-. As we said, we will not deal with them in this paper (see [3, 6] for more on this topic).

At this point, papers on explicit substitutions typically present a long list of σ-reductions aimed at confining substitution application and composition to specific positions in a term and a substitution,

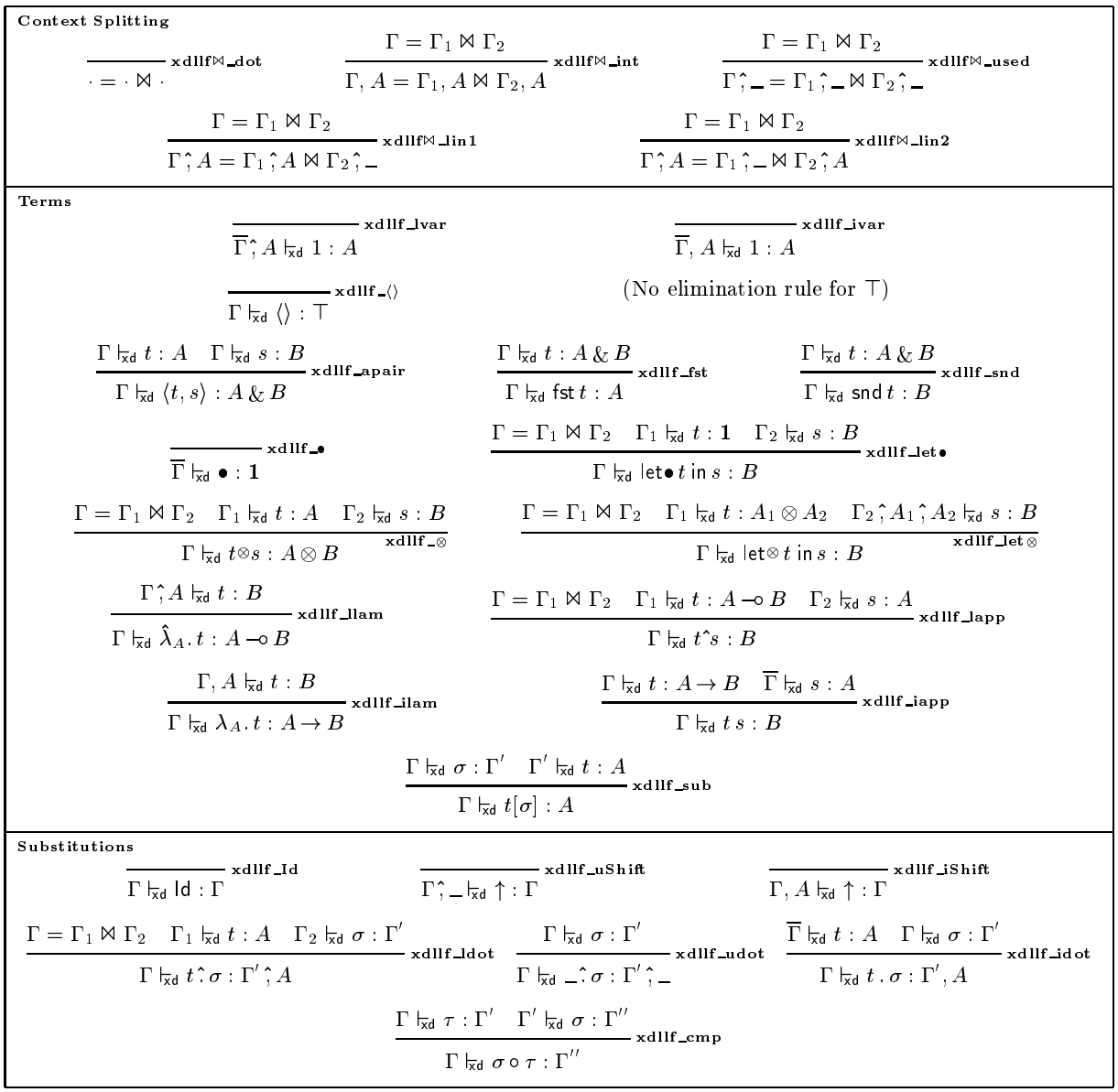

Figure 2: Typing in xdLLF$^-$

respectively. Their systematic application yields σ-normal forms. This is very convenient as σ-rules can be interleaved with other reductions to efficiently reach canonical forms.

Unfortunately, linear explicit substitutions are not as cooperative. Indeed, pushing a substitution τ through a multiplicative operator typically requires splitting it non-deterministically. This may be difficult unless τ is in σ-normal form. Moreover, not every split is sound: constraints (e.g. typing information)

are needed to guarantee correctness. In [6], we avoided unsound splits by checking for free variables and splitting the substitution accordingly. A direct implementation of this approach requires either to collect them by visiting every subterm of a binary multiplicative operator, or, less naively, to maintain a list of free linear variables for every such term. This is not satisfactory since both techniques involve a significant overhead. Furthermore, checking for free variables does not scale up to languages with meta-variables (needed in linear logical frameworks and higher-order linear logic programming).

The approach we propose here replaces σ-rules with a declarative specification of how to transform an arbitrary typable xdLLF$^-$ term or substitution into an equivalent object in σ-normal form. This corresponds to using a big-step as opposed to a small-step semantics in programming language theory. We handle substitution splitting by σ-normalizing substitutions before applying them, and by using a typing derivation to guide the splitting process. Although this approach deals correctly with linearity, it still has several drawbacks. First, it fixes the σ-reduction strategy, which is instead left open when σ-rules are used. Second, it does not allow interleaving σ-normalization steps with other reductions. Third, as for the proposal in [6], it does not scale up to handle meta-variables. Nonetheless, we see it as a valuable first step toward addressing these issues more satisfactorily.

4 σ-Normalization

In a σ-normal term, every substitution application has the form $1[\uparrow \circ \ldots \circ \uparrow]$. If there are j instances of $\uparrow$, this effectively implements the de Bruijn index $j+1$. Similarly, σ-normal substitutions consist of a list of linear or intuitionistic σ-normal substitution-terms terminated by the composition of a number of $\uparrow$. They correspond to environments in a functional programming setting.

We formalize this description as a language that we call xdLLF$^-_\sigma$:

σ-normal terms:

$$
\begin{array}{lllll}
u ::= & i & & & \\
 & \mid \langle\rangle & & & \\
 & \mid \langle u_1, u_2\rangle & \mid \mathsf{fst}\, u & \mid \mathsf{snd}\, u & \\
 & \mid \bullet & \mid \mathsf{let}\bullet\, u_1\, \mathsf{in}\, u_2 & & \\
 & \mid u_1 \otimes u_2 & \mid \mathsf{let}\otimes\, u_1\, \mathsf{in}\, u_2 & & \\
 & \mid \hat{\lambda}_A.\, u & \mid u_1 \hat{} u_2 & & \\
 & \mid \lambda_A.\, u & \mid u_1\, u_2 & &
\end{array}
$$

σ-normal substitutions:

$$
\begin{array}{lll}
\rho ::= & \uparrow^j & (\textit{shifts}) \\
 & \mid u \hat{} \rho & (\textit{linear extension}) \\
 & \mid \_ \hat{} \rho & (\textit{used linear extension}) \\
 & \mid u \,.\, \rho & (\textit{intuitionistic extension})
\end{array}
$$

where i is a positive integer, 1, 2, 3, ... (de Bruijn index) and j is a non-negative integer ($\uparrow^0$, $\uparrow^1$, $\uparrow^2$, etc. are all legal substitutions). Types and contexts are as in xdLLF$^-$. So are their operations. The typing rules for xdLLF$^-_\sigma$ are given in Figure 3. Reductions are adapted from xdLLF$^-$.

We recover xdLLF$^-$ terms and substitutions by rewriting i as $1[\uparrow^{i-1}]$, and then recursively expanding $\uparrow^j$ to the composition of j shifts in an xdLLF$^-_\sigma$ term or substitution. It can be shown that this translation preserves typing and reductions [3].

The σ-normalization procedure we propose is partially displayed in Figure 4. It is articulated in a number of judgments of the form $L \longrightarrow\!\!\!\!\!\rightarrow R$ where the right-hand side R is a σ-normal term (or substitution), while the left-hand side L is an xdLLF$^-$ term (resp. substitution) in the zones marked "congruences", or a σ-normal substitution ρ_0 applied to an xdLLF$^-$ term (resp. composed with an xdLLF$^-$ substitution) in the zones labeled "reductions". We rely on several auxiliary operations and judgments. The intuitionistic part $\overline{\rho}$ of a σ-normal substitution ρ is obtained by replacing every linear substitution term in ρ with "_",

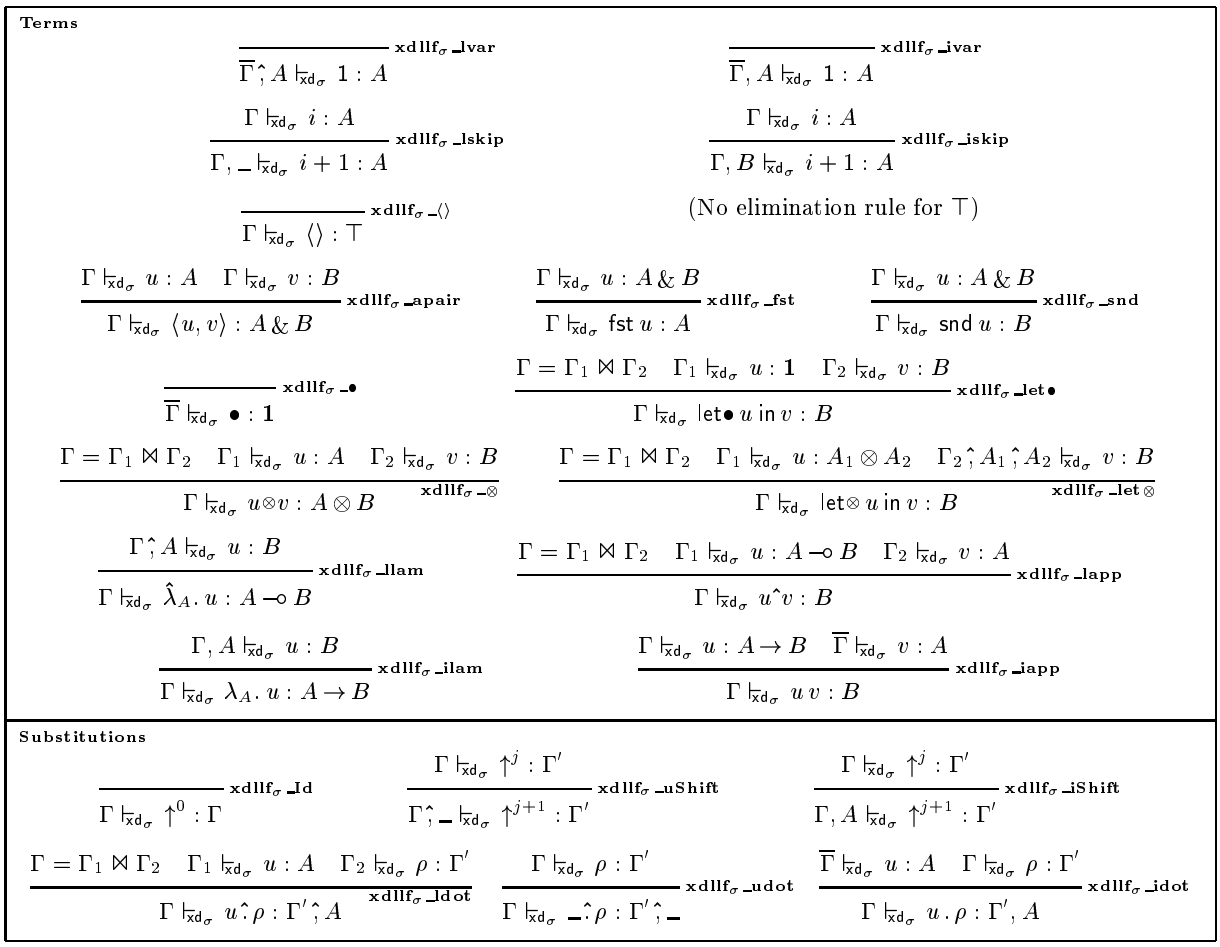

Figure 3: Typing in xdLLF^-_σ

as for contexts. The substitution splitting judgment $\rho = \rho_1 \bowtie \rho_2$ is also defined similarly to the analogous relation on contexts; notice that it operates only on xdLLF^-_σ substitutions; observe also that, like context splitting, this operation is non-deterministic (e.g. $u\hat{.}\uparrow^0$ can be split as either $u\hat{.}\uparrow^0 = u\hat{.}\uparrow^0 \bowtie \_\hat{.}\uparrow^0$ or $u\hat{.}\uparrow^0 = \_\hat{.}\uparrow^0 \bowtie u\hat{.}\uparrow^0$). Finally, we have the linear and intuitionistic binder crossing operations: $\hat{+}(\rho)$ and $+(\rho)$, and the associated reduction judgment (not displayed). They are what it takes to push the substitution ρ through a binder: intuitively, they correspond to $1\hat{.}(\rho \circ \uparrow)$ and $1\,.\,(\rho \circ \uparrow)$, respectively. These omitted definitions can be found in [3].

The procedure in Figure 4 works as follows: it walks through an xdLLF^- term using the congruence rules till the first substitution application $t[\tau]$ is encountered. Then, it normalizes τ to ρ_0 and switches to the reduction rules to push ρ_0 inside t. This recursive descent terminates when constructors without arguments are processed.

This procedure terminates since each constructor in an xdLLF^- term or substitution is visited exactly once by the congruence or reduction judgments. However, it can fail unless it is guided by a typing

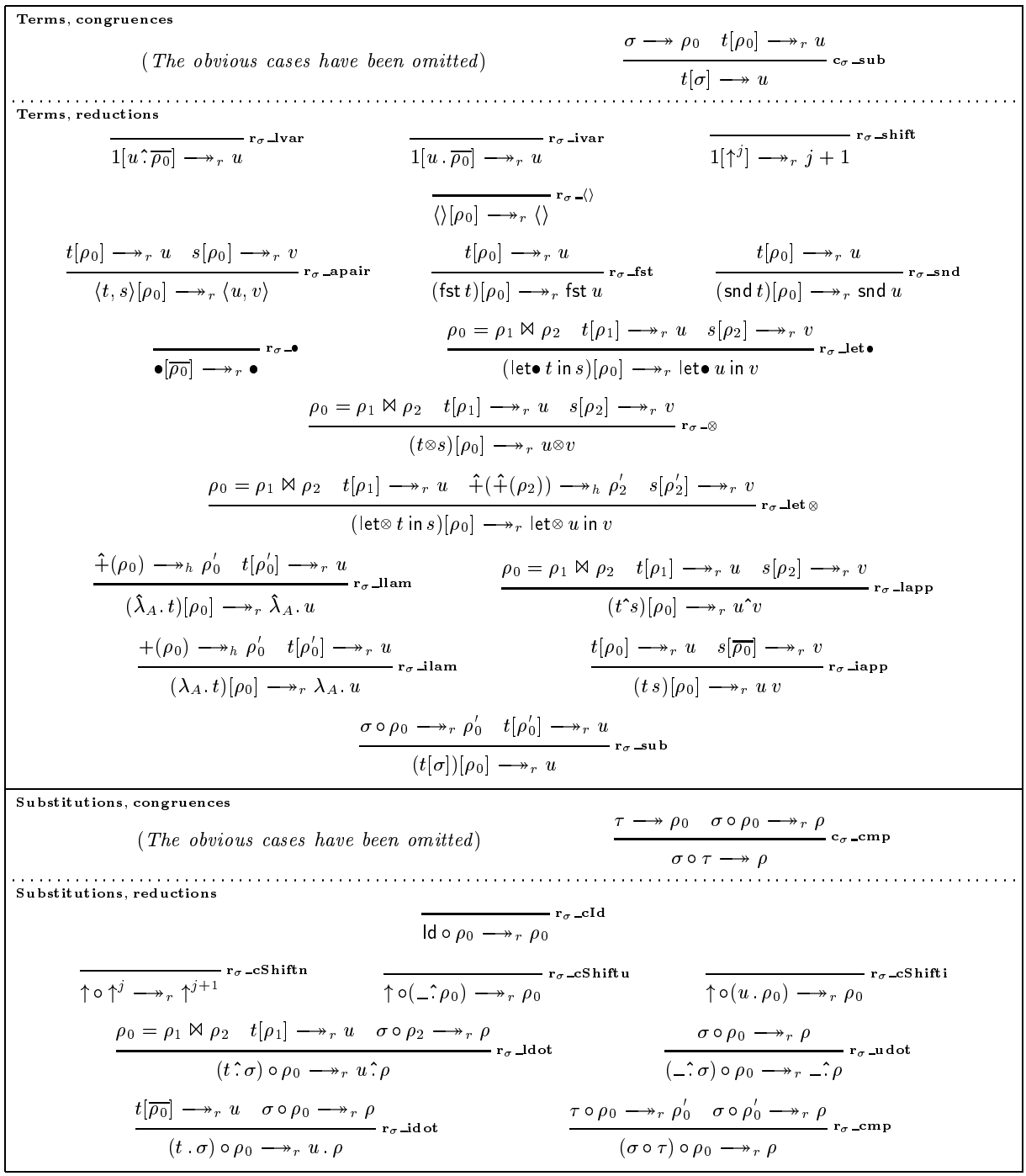

Figure 4: σ-Normalization of xdLLF$^-$ Terms and Substitutions

derivation, even when starting from a typable term (or substitution). Indeed, objects such as $\overline{\rho_0}$ in rule $\mathbf{r}_{\sigma\_\bullet}$ (Figure 4) act as constraints on the terms (or substitutions) that can be acceptably processed. More precisely, in a given context only a few (just one in the absence of $\langle\rangle$) of the choices for splitting a substitution will satisfy these constraints, namely the ones which split the substitution according to the linear variables used in the corresponding terms. The typing rules of $\mathrm{xdLLF}^{-}_{\sigma}$ ensure that only those splits can be used for reduction. The substitution splitting judgment can generate substitutions that will not pass this test, unless properly controlled. However, no problem arises when σ-reduction is supervised by a typing derivation, as stated by the following theorem. (We abbreviate "*$\mathcal{D}$ is a derivation of judgment J*" as $\mathcal{D} :: J$.)

Theorem 4.1 (*Subject reduction for* $\longrightarrow\!\!\!\!\!\rightarrow$)

i. If $\mathcal{T} :: \Gamma \vdash_{\overline{\mathsf{xd}}} t : A$, *then there exist* u, $\mathcal{R}$, *and* $\mathcal{U}$ *such that* $\mathcal{R} :: t \longrightarrow\!\!\!\!\!\rightarrow u$ *and* $\mathcal{U} :: \Gamma \vdash_{\overline{\mathsf{xd}}_\sigma} u : A$.

ii. If $\mathcal{S} :: \Gamma \vdash_{\overline{\mathsf{xd}}} \sigma : \Gamma'$, *then there exist* ρ, $\mathcal{R}$, *and* $\mathcal{V}$ *such that* $\mathcal{R} :: \sigma \longrightarrow\!\!\!\!\!\rightarrow \rho$ *and* $\mathcal{V} :: \Gamma \vdash_{\overline{\mathsf{xd}}_\sigma} \rho : \Gamma'$.

Proof
After proper generalization of the statement, the proof proceeds by structural induction on $\mathcal{S}$ and $\mathcal{T}$. It relies on several auxiliary results omitted for space reasons. See [3] for details. ☑

We expect our σ-normalization procedure to be a function (even when no accompanying typing derivation is given), but we have not had the time yet to verify this conjecture. Further results about this procedure can be found in [3].

5 Future Developments

We have described the linear λ-calculus with explicit substitutions xdLLF^{-}, which appears to be a reasonable playground for experimenting with linear explicit substitutions as it isolates some of the most delicate interactions between explicit substitutions and linearity. This paper analyzed one approach to coping with them, although it has several drawbacks that make it unsuitable as a foundational calculus for a linear logical framework: in particular, it cannot be directly extended to handle meta-variable (essential for logical framework implementations), and it does not allow interleaving σ-reductions with other reduction steps (making it very rigid). Nonetheless, we see it as a valuable first step toward addressing these issues more satisfactorily. A few immediate questions arise from this endeavor:

- What happens if we use dual contexts as in xDILL?
- What if we make substitutions dual too?
- How do named variables influence the result (see also [7] on this)?
- Do we still get correct σ-normal terms if we omit splitting substitutions?
- Can we define a usable notion of weak σ-normal forms?
- How do meta-variables fit into all this?

Answers to some of these questions should prove highly relevant to the implementation of linear logical frameworks, linear functional languages and other systems based on linear λ-calculi. We are currently experimenting with languages that incorporate each of these ideas.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Andrew Barber. *Linear Type Theories, Semantics and Action Calculi*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1997.

[3] Iliano Cervesato, Valeria de Paiva, and Eike Ritter. Explicit substitution for linear logical frameworks. Unpublished manuscript, `http://www.stanford.edu/~iliano/papers/Forthcoming/xdllf.ps.gz`.

[4] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science — LICS'96*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[5] N.G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[6] Neil Ghani, Valeria de Paiva, and Eike Ritter. Linear explicit substitutions. In *Proceedings of the First International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs — WESTAPP'98*, Tsukuba, Japan, March 1998.

[7] Eike Ritter and Valeria de Paiva. On explicit substitution and names (extended abstract). In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 248–258. Springer-Verlag LNCS 1256, 1997.