Proofs as Terms, Terms as Graphs

Jui-Hsuan Wu[0000−0001−5880−5379]

LIX, Institut Polytechnique de Paris, Palaiseau, France
jwu@lix.polytechnique.fr

Abstract. Starting from an encoding of untyped λ-terms with sharing, defined using synthetic inference rules based on a focused proof system for Gentzen's *LJ*, we introduce the positive λ-calculus, a call-by-value calculus with explicit substitutions. This calculus is closely related to Accattoli and Paolini's value substitution calculus but has a different style of reduction rules that provides a good notion of sharing along the reduction. We also propose a graphical representation in order to capture the structural equivalence on terms that can be described using rule permutations. On one hand, this graphical representation provides a way to remove redundancy in the syntax, and on the other hand, it allows implementing basic operations such as substitution and reduction in a straightforward way.

Keywords: Proof theory · Term representation · λ-calculus · Sharing · Graphical representation

1 Introduction

Terms (or expressions) are essential in different settings: mathematical proofs, programming languages, proof assistants, etc. To prevent redundancy in these systems, a canonical and compact syntactic structure is needed. Proof theory has been broadly used in the studies of term representation. There have been several different approaches to address the question of the canonicity of proofs, such as proof nets [20], expansion trees [25], focusing [7], combinatorial proofs [22], etc.

Focusing and synthetic inference rules. In this paper, we choose to use focusing as our main tool. Andreoli introduced the first focused proof system to describe proofs in linear logic in a more structured way. Inference rules are organized into two different phases: *positive* and *negative* phases. Andreoli [8] and Chaudhuri [11] suggested that phases should be viewed as large-scale rules for proof construction. In [24], Marin et al. defined another version of large-scale rules, called *synthetic inference rule*, which is essentially composed of a negative phase and a positive phase, in order to provide more high-level descriptions of proof systems. The invention of synthetic inference rules also provides a systematic way to extend proof systems such as *LJ* and *LK*.

Various focused proof systems have been proposed for *LJ* [21,18,23] and *LK* [15,23]. Several authors also designed term calculi for *LJT* [21], *LJQ* [18], and *LJF* [10]. In [27], Miller and Wu use synthetic inference rules built using the focused proof system *LJF* to study term structures. In *LJF*, formulas are polarized and different polarity assignments yield different forms of proofs and thus provide different styles of term representation.

In this paper, we are interested in their encoding of untyped λ-terms defined by giving the positive polarity to every atomic formula. Unlike the usual syntax of untyped λ-terms, this encoding constructs terms in a *bottom-up* style and allows sharing within a term using *named structures*, or *explicit substitutions*. This term representation is, however, not compact, in the sense that a lot of named structures can be permuted or put in parallel. These permutations correspond, in fact, to phase permutations in focused proof systems, and to rule permutations in terms of synthetic inference rules.

Focusing and graphical structures. Several authors have proposed *multi-focused* proof systems to illustrate this phenomenon for *MALL* [26,16,13], *LK* [12], and *LJ* [28]. Moreover, in each of [13] and [12], an isomorphism has been established between the multi-focused proof system and a graphical representation of proofs (proof nets and expansion trees, respectively). This is where our *graphical representation* for terms comes in. Though inspired by these works on multi-focusing, we choose to put our focus on terms, i.e. (single-)focused proofs, and define the structural equivalence that can also be justified by rule permutations. Also, note that the structural equivalence considered in this paper does not cover all the possible permutations that are captured by multi-focused proof systems.

Explicit substitution and proof nets. Historically, there have been several attempts to connect λ-calculus with sharing, or explicit substitution [1], to proof nets. This connection was first proposed by Di Cosmo and Kesner [17] to study cut-elimination. Recently, proof nets have also been connected to calculi such as the call-by-value λ-calculus [2] and the linear substitution calculus [3]. In this paper, like in each of [2] and [3], we show that the structural equivalence on terms is exactly the same as the one captured by the graphical representation. In contrast to these works that often introduce boxes to deal with sharing, we decide to use the notion of *bodies*, which is closely related to the notion of *level* in these works. This choice does not make a huge difference in our theoretical results, but it provides a clear way to establish the correspondence between the usual syntax and the graphical one.

Graphs and sharing. Graphs have been widely used in the studies of sharing. Since Wadsworth [29], several authors have proposed different graphical structures to study full laziness (see [9] for an overview), a concept related to evaluation based on the call-by-need mechanism. Another aspect about sharing, called *sharing equality*, whose goal is to decide whether the unfoldings of two terms with sharing are equal, has also been discovered in recent years. A linear algorithm was proposed in [14] based on a graphical representation called *λ-graphs* for untyped λ-terms. This graphical representation is close to the one studied in our work, which allows us to apply the algorithm without any difficulty.

Contribution.

1. We propose a rewrite system for the encoding of untyped λ-terms proposed in [27], called the *positive λ-calculus*. Note that it is *inspired* by proof theory but does not follow the usual ”redex = cut” paradigm. This rewrite system follows a call-by-

value discipline and is closely related to the value substitution calculus [6] and the linear substitution calculus [4].

2. We propose a graphical representation, called λ-graphs with bodies, for this encoding of untyped λ-terms and provide a one-to-one correspondence between λ-graphs with bodies and terms up to permutations of independent named structures (Theorem 5).
3. We describe how substitution and reduction can be easily implemented on λ-graphs with bodies.

Proofs. Some proofs are ignored and can be found in Appendix.

2 Preliminaries: the focused proof system LJF and synthetic inference rules

Fix a set ATOM of *atomic formulas*. *Formulas* are built with implications and atomic formulas. An *atomic bias assignment* is a map δ from ATOM to $\{+,-\}$. A *polarized formula* (resp. *polarized theory*) is a formula (resp. multiset of formulas) together with an atomic bias assignment. Implications are negative and atomic formulas can be either positive or negative following the atomic bias assignment. In *LJF*, there are two kinds of sequents: $\Uparrow$-sequents $\Gamma \Uparrow \Theta \vdash \Delta \Uparrow \Delta'$ and $\Downarrow$-sequents $\Gamma \Downarrow \Theta \vdash \Delta \Downarrow \Delta'$. Here, $\Gamma, \Theta, \Delta, \Delta'$ are multisets of formulas. Γ and Δ' are called *storage zones* and Θ and Δ are called *staging zones*. In a $\Downarrow$-sequent, exactly one of Θ and Δ can be non-empty and contains exactly one formula. To simplify the notation, we drop an arrow whenever its corresponding staging zone is empty. As a result, sequents without any arrow are $\Uparrow$-sequents as any $\Downarrow$-sequent has exactly one non-empty staging zone. Furthermore, $\Delta \cup \Delta'$ is a singleton, as we are in an intuitionistic setting. The rules of the implicational fragment of *LJF* are presented in Figure 1. *LJF* proofs have a two-phase structure: $\Uparrow$-phases and $\Downarrow$-phases.

Decide, Release, and Store Rules

$$\frac{N, \Gamma \Downarrow N \vdash A}{N, \Gamma \vdash A}\ D_l \quad \frac{\Gamma \vdash P \Downarrow}{\Gamma \vdash P}\ D_r \quad \frac{\Gamma \Uparrow P \vdash A}{\Gamma \Downarrow P \vdash A}\ R_l \quad \frac{\Gamma \vdash N \Uparrow}{\Gamma \vdash N \Downarrow}\ R_r$$

$$\frac{\Gamma, C \Uparrow \Theta \vdash \Delta \Uparrow \Delta'}{\Gamma \Uparrow \Theta, C \vdash \Delta \Uparrow \Delta'}\ S_l \quad \frac{\Gamma \Uparrow \Theta \vdash A}{\Gamma \Uparrow \Theta \vdash A \Uparrow}\ S_r$$

Initial Rules

$$\frac{\delta(A) = +}{A, \Gamma \vdash A \Downarrow}\ I_r \quad \frac{\delta(A) = -}{\Gamma \Downarrow A \vdash A}\ I_l$$

Introduction Rules for implication

$$\frac{\Gamma \vdash B \Downarrow \qquad \Gamma \Downarrow B' \vdash A}{\Gamma \Downarrow B \supset B' \vdash A} \supset L \quad \frac{\Gamma \Uparrow \Theta, B \vdash B' \Uparrow}{\Gamma \Uparrow \Theta \vdash B \supset B' \Uparrow} \supset R$$

Fig. 1. The implicational fragment of the focused proof system *LJF*. Here, P is positive, N is negative, A is atomic, and B, B' and C are arbitrary formulas.

Definition 1 (Synthetic inference rules, [24]). *A synthetic inference rule is a rule of the form*

$$\frac{\Gamma_1 \vdash A_1 \quad \cdots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A}\, N \qquad \text{justified } \textit{by an LJF derivation of the form} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash A_1 \quad \cdots \quad \Gamma_n \vdash A_n \\ \Pi \\ \Gamma \Downarrow N \vdash A\end{array}}{\Gamma \vdash A}\, D_l$$

Here, N is a negative formula that appears in Γ, $n \geq 0$, and within Π, a $\Downarrow$-sequent never occurs above an $\Uparrow$-sequent. The structure of LJF proofs implies $N \in \Gamma_i$ for all $1 \leq i \leq n$. This rule is called the synthetic inference rule for N.

$$\dfrac{\dfrac{}{\Gamma \vdash D \Downarrow}\, I_r \qquad \dfrac{\dfrac{}{\Gamma \vdash D \Downarrow}\, I_r \qquad \dfrac{\Gamma, D \vdash D}{\Gamma \Downarrow D \vdash D}\, S_l/R_l}{\Gamma \Downarrow D \supset D \vdash D}\, \supset L}{\dfrac{\Gamma \Downarrow D \supset D \supset D \vdash D}{\Gamma \vdash D}\, D_l}\, \supset L$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, D \vdash D}{\Gamma \Uparrow D \vdash D \Uparrow}\, S_l/S_r}{\Gamma \vdash D \supset D \Uparrow}\, \supset R}{\Gamma \vdash D \supset D \Downarrow}\, R_r \qquad \dfrac{\Gamma, D \vdash D}{\Gamma \Downarrow D \vdash D}\, S_l/R_l}{\Gamma \Downarrow (D \supset D) \supset D \vdash D}\, \supset L}{\Gamma \vdash D}\, D_l$$

Fig. 2. Two derivations justifying synthetic inference rules (D is atomic and polarized positively).

Synthetic inference rules show how a formula can be *used* in a proof and they can be used to extend sequent systems such as *LJ*. Since the synthetic inference rules for atomic formulas are exactly the same as the initial rule of *LJ*, we will only consider synthetic inference rules for non-atomic formulas. We give in Figure 2 two *LJF* derivations justifying synthetic inference rules for two different formulas. In [24], it is shown that the synthetic inference rules for certain formulas are particularly simple and can be easily used to extend *LJ* with a (polarized) theory. To express this, we first define the order of a formula.

Definition 2. *The order of a formula B, written $ord(B)$, is defined as follows: $ord(A) = 0$ if A is atomic and $ord(B_1 \supset B_2) = max(ord(B_1) + 1, ord(B_2))$.*

Definition 3 (Extensions of *LJ* by polarized theories, [24]). *Let $\mathcal{T}$ be a finite multiset of formulas of order one or two, and let δ be an atomic bias assignment. We define the extension $LJ\lfloor\mathcal{T},\delta\rfloor$ of LJ by the polarized theory $(\mathcal{T},\delta)$ to be the two-sided proof system built as follows. The only sequents in the $LJ\lfloor\mathcal{T},\delta\rfloor$ proof system are of the form $\Gamma \vdash A$ where A is atomic and Γ is a multiset of atomic formulas. The inference rules of $LJ\lfloor\mathcal{T},\delta\rfloor$ include the initial rule of LJ and for every synthetic inference rule*

$$\frac{N, \Gamma_1 \vdash A_1 \qquad \ldots \qquad N, \Gamma_n \vdash A_n}{N, \Gamma \vdash A}\, N$$

where $N \in \mathcal{T}$, then the rule

$$\frac{\Gamma_1 \vdash A_1 \qquad \ldots \qquad \Gamma_n \vdash A_n}{\Gamma \vdash A}\, N$$

$$\frac{}{\Gamma \vdash x : D}\ nvar \quad x : D \in \Gamma$$

$$\frac{\Gamma \vdash s : D \qquad \Gamma \vdash t : D}{\Gamma \vdash st : D}\ napp$$

$$\frac{\Gamma, x : D \vdash t : D}{\Gamma \vdash \lambda x.t : D}\ nabs$$

$$\frac{}{\Gamma \vdash x : D}\ pvar \quad x : D \in \Gamma$$

$$\frac{\Gamma, x : D \vdash t : D}{\Gamma \vdash t[x \leftarrow yz] : D}\ papp \quad \{y : D, z : D\} \subseteq \Gamma$$

$$\frac{\Gamma, y : D \vdash s : D \qquad \Gamma, x : D \vdash t : D}{\Gamma \vdash t[x \leftarrow \lambda y.s] : D}\ pabs$$

Fig. 3. Annotated inference rules of the systems $LJ\lfloor\Gamma_0, \delta^-\rfloor$ (left) and $LJ\lfloor\Gamma_0, \delta^+\rfloor$ (right) where $\delta^-(D) = -$, $\delta^+(D) = +$ and $\Gamma_0 = \{D \supset D \supset D, (D \supset D) \supset D\}$.

is included in $LJ\lfloor\mathcal{T}, \delta\rfloor$.

Note that in the original version proposed in [24], $\mathcal{T}$ is a set instead of a multiset. In fact, each formula in $\mathcal{T}$ corresponds to a combinator in annotated proofs. Different occurrences of the same formula define different combinators. This is why we consider multisets instead of sets of formulas.

The following theorem justifies Definition 3.

Theorem 1. *Let $\mathcal{T}$ be a finite multiset of formulas of order one or two, and let δ be an atomic bias assignment. Then for any atomic formula A and Γ containing atomic formulas only, $\Gamma, \mathcal{T} \vdash A$ is provable in LJ if and only if $\Gamma \vdash A$ is provable in $LJ\lfloor\mathcal{T}, \delta\rfloor$.*

3 The positive λ-calculus

The notion of terms, which is usually a primitive notion in most of the literature, is a derived notion here: they are annotations of proofs. By annotating inference rules in the proof system $LJ\lfloor\mathcal{T}, \delta\rfloor$, we obtain rules for various combinators, each one of which corresponds to a formula in $\mathcal{T}$. In [27], Miller and Wu give an encoding of untyped λ-terms by considering the theory $\Gamma_0 = \{D \supset D \supset D, (D \supset D) \supset D\}$ where D is an atomic formula. If D is given the negative polarity, we get the *negative bias syntax*, i.e., the usual tree structure for untyped λ-terms. If D is given the positive polarity, we get the *positive bias syntax*, a structure where explicit sharing is possible. The annotated inference rules obtained using both polarity assignments are shown in Figure 3. As an example, the *papp* and *pabs* rules are obtained from the two derivations given in Figure 2. Note that the branches ended with I_r rule become a side condition on the schema variable Γ.

In the following, we introduce a calculus called *positive λ-calculus* based on terms built using the positive bias syntax. Since the sequents and inference rules considered only involve the atomic formula D, we often replace the annotated formula $x : D$ (resp. $t : D$) with simply its annotation x (resp. t).

Terms. Fix a set NAME $= \{x, y, z, \ldots\}$ of *names* (or *variables*). The set TERM of terms, denoted by $s, t, u, \ldots$, is generated by the following grammar:

$$\text{Terms} \quad s, t \coloneqq x \mid t[x \leftarrow yz] \mid t[x \leftarrow \lambda y.s]$$

A term is essentially a list of constructs $[x \leftarrow p]$, called *explicit substitutions* or *named structures*, preceded by a name, which we call the *output* of the term.

The set $fv(t)$ of *free variables* of a term t is given by: $fv(x) = \{x\}$, $fv(t[x \leftarrow yz]) = (fv(t) \setminus \{x\}) \cup \{y, z\}$ and $fv(t[x \leftarrow \lambda y.s]) = (fv(s) \setminus \{y\}) \cup (fv(t) \setminus \{x\})$. Note that every name x introduced by the construct $[x \leftarrow p]$ is bound. We should consider terms up to α-equivalence and assume that all bound names are distinct from each other and from any free variables.

A *signature* is a finite subset Σ of NAME. We write Σ, x for $\Sigma \cup \{x\}$ and this also implies that $x \notin \Sigma$. We say that t is a *Σ-term* if $\Sigma \vdash t$ using the *pvar*, *papp*, and *pabs* rules in Figure 3.

Contexts. We define *contexts* and *left contexts* by the following grammar:

$$\begin{array}{ll} \textsc{Contexts} & C \coloneqq \square \mid C[x \leftarrow yz] \mid C[x \leftarrow \lambda y.s] \mid t[x \leftarrow \lambda y.C] \\ \textsc{Left Contexts} & L \coloneqq \square \mid L[x \leftarrow yz] \mid L[x \leftarrow \lambda y.s] \end{array}$$

The *plugging* $C\langle t\rangle$ (resp. $L\langle t\rangle$) of t in the context C (resp. left context L) is obtained from C (resp. L) by replacing the placeholder $\square$ with t. Every term can be written uniquely as $L\langle x\rangle$ for some left context L and variable x. Note that we allow plugging in a context to capture variables. A *congruence* on terms is an equivalence relation that is closed by context.

Structural equivalence. Named structures can be seen as intermediate definitions within a term. If two definitions are independent of each other, we should be able to permute them. By defining $fv(yz) = \{y, z\}$ and $fv(\lambda y.s) = fv(s) \setminus \{y\}$, this can be expressed using the equation:

$$t[x_1 \leftarrow p_1][x_2 \leftarrow p_2] \sim_{\mathtt{str}} t[x_2 \leftarrow p_2][x_1 \leftarrow p_1] \qquad \text{if } x_1 \notin fv(p_2) \text{ and } x_2 \notin fv(p_1)$$

Definition 4 (Structural equivalence). *We define an equivalence relation $\equiv_{\mathtt{str}}$ on terms, called the structural equivalence, as the smallest congruence containing $\sim_{\mathtt{str}}$.*

Note that the structural equivalence can also be justified by rule permutations.

Substitution. In [27], there is a big-step (atomic) cut-elimination procedure for proofs built using synthetic inference rules. This procedure provides a definition of substitution for terms.

Definition 5 (Substitution on terms, [27]). *Let t, u be terms and x a name such that $x \notin fv(u)$. We define the result of substituting u for x in t, written $t[x/u]$, as follows:*

$$t[x/s[y \leftarrow zw]] = t[x/s][y \leftarrow zw] \qquad t[x/s[y \leftarrow \lambda z.u]] = t[x/s][y \leftarrow \lambda z.u]$$

Here, $t[x/y]$ is obtained from t by renaming x to y. Note that by expressing the term u uniquely as $L\langle y\rangle$, we have $t[x/u] = L\langle t[x/y]\rangle$ by a straightforward induction.

Unfolding and reduction. Term reduction is often related to cut-elimination in the literature. However, terms considered here correspond to cut-free proofs. A natural question to ask is: How should we evaluate them? Of course, we could *unfold* all the named structures of a term and get its corresponding untyped λ-term.

Top-level Rules

$$C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.L\langle y'\rangle] \mapsto_{\mathtt{beta}} C\langle L\langle t\{z/y'\}\rangle\{y/w\}\rangle[x \leftarrow \lambda y.L\langle y'\rangle]$$
$$t[x \leftarrow \lambda y.s] \mapsto_{\mathtt{gc}} t \qquad \text{if } x \notin fv(t)$$

Rewrite Rules

$$C\langle t\rangle \to_{\mathtt{beta}} C\langle u\rangle \qquad \text{if } t \mapsto_{\mathtt{beta}} u$$
$$C\langle t\rangle \to_{\mathtt{gc}} C\langle u\rangle \qquad \text{if } t \mapsto_{\mathtt{gc}} u$$
$$t \to_{\mathtt{pos}} u \qquad \text{if } t \to_{\mathtt{beta}} u \text{ or } t \to_{\mathtt{gc}} u$$

Fig. 4. Rewrite rules of the positive λ-calculus.

Definition 6. *The unfolding $\underline{t}$ of a term t is the untyped λ-term defined as follows:*

$$\underline{x} = x \qquad \underline{t[x \leftarrow yz]} = \underline{t}\{x/yz\} \qquad \underline{t[x \leftarrow \lambda y.s]} = \underline{t}\{x/\lambda y.\underline{s}\}$$

where $\{\cdot/\cdot\}$ is the meta-level substitution of untyped λ-terms.

For example, we have $\underline{y[y \leftarrow fz][f \leftarrow \lambda x.x]} = (\lambda x.x)z$. This definition provides a way to translate from the positive bias syntax to the negative bias syntax. Note that the unfolding of a term is not necessarily a β-normal untyped λ-term. For a term t, we could refer to the β-normal form of its unfolding as its *meaning*. However, computing unfoldings of terms might require exponential costs.

Therefore, we proceed in a different way here: we look for a reduction procedure that only involves the positive bias syntax and is compatible with the β-reduction in the negative bias syntax (a reduction step should not change the meaning of a term). We now define `beta`-*redexes* and the `beta`-*rule*. Consider the following annotated proof:

$$\dfrac{\begin{array}{c}\Pi_1 \\ \Sigma, y : D \vdash s : D\end{array} \qquad \begin{array}{c}\dfrac{\begin{array}{c}\Pi_2 \\ \Sigma', x : D, z : D \vdash t : D\end{array}}{\Sigma', x : D \vdash t[z \leftarrow xw] : D}\ papp \\ \vdots \\ \Sigma, x : D \vdash C\langle t[z \leftarrow xw]\rangle : D\end{array}}{\Sigma \vdash C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.s] : D}\ pabs$$

with $w : D \in \Sigma', x : D$ and C a context. In the term annotating the conclusion, the name x is used to introduce an abstraction $\lambda y.s$ and is later applied to an argument w. We call the named application pattern xw here a `beta`-*redex*. To eliminate this `beta`-redex, we shall consider the following proof:

$$\dfrac{\begin{array}{c}\Pi_1' \\ \Sigma', x : D \vdash s\{y/w\} : D\end{array} \qquad \begin{array}{c}\Pi_2 \\ \Sigma', x : D, z : D \vdash t : D\end{array}}{\Sigma', x : D \vdash Cut(z.t, s\{y/w\}) : D}\ cut$$

where Π_1' is obtained from Π_1 by variable renaming and weakening. By eliminating this cut, we obtain a cut-free proof of the conclusion $\Sigma', x : D \vdash t[z/s\{y/w\}] : D$. This gives

the following `beta`-*rule*:

$$C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.s] \mapsto_{\texttt{beta}} C\langle t[z/s\{y/w\}]\rangle[x \leftarrow \lambda y.s]$$

that can also be expressed using left contexts as shown in Figure 4. We also consider a garbage collection rule for named abstractions. This rule can be justified by the fact that if a formula is never used in a proof, then we can remove the rule that introduces it.

Intuitively, to eliminate a `beta`-redex, it suffices to make a copy of the abstraction body, make a variable renaming within this copy and a variable renaming in the rest of the term. We illustrate these steps in the following example:

$$\begin{array}{ll} x_2[x_2 \leftarrow f x_1][x_1 \leftarrow f x_0][f \leftarrow \lambda x.z[z \leftarrow yy][y \leftarrow xx]] & \to_{\texttt{beta}} \\ x_2[x_2 \leftarrow f z_1][z_1 \leftarrow y_1 y_1][y_1 \leftarrow x_0 x_0][f \leftarrow \lambda x.z[z \leftarrow yy][y \leftarrow xx]] & \to_{\texttt{beta}} \\ z_2[z_2 \leftarrow y_2 y_2][y_2 \leftarrow z_1 z_1][z_1 \leftarrow y_1 y_1][y_1 \leftarrow x_0 x_0][f \leftarrow \lambda x.z[z \leftarrow yy][y \leftarrow xx]] & \to_{\texttt{gc}} \\ z_2[z_2 \leftarrow y_2 y_2][y_2 \leftarrow z_1 z_1][z_1 \leftarrow y_1 y_1][y_1 \leftarrow x_0 x_0] & \end{array}$$

Like the VSC, the positive λ-calculus enjoys the confluence property.

Theorem 2. *The positive λ-calculus is confluent.*

The $\to_{\texttt{pos}}$ is not terminating as shown by the term $w[w \leftarrow xx][x \leftarrow \lambda y.z[z \leftarrow yy]]$. The following proposition shows that $\to_{\texttt{pos}}$ does not affect the meaning of a term.

Proposition 1. *Let s and t be terms such that $s \to_{\texttt{pos}} t$. Then $\underline{s} \to_\beta^* \underline{t}$.*

It is easy to see that, for every named application $[x \leftarrow yz]$ in a normal term t, y is not a name introducing an abstraction. We have thus the following proposition.

Proposition 2. *If s is a normal term, then $\underline{s}$ is β-normal.*

The converse is however not true as shown by the term $t = z'[z \leftarrow wx][x \leftarrow \lambda y.s]$. $\underline{t} = z'$ is β-normal but t is normal if and only if s is normal.

This rewrite system follows a call-by-value discipline that can be observed in the example above as there is no way to remove the named application $[z \leftarrow wx]$. Also, note that it is a *strong* calculus as we allow reduction under an abstraction.

The positive λ-calculus is closely related to the value substitution calculus (VSC) of [6], presented in Figure 5, and the linear substitution calculus (LSC) of [4], presented in Figure 6. In both systems, `m` stands for multiplicative and `e` stands for exponential (these terms come from the literature on linear logic). On one hand, the positive λ-calculus has a call-by-value behavior similar to that of the VSC, and on the other hand, it admits a micro-step exponential rule (for named abstractions) as in the LSC. Another difference between the positive λ-calculus and the VSC is that in the positive λ-calculus, a named abstraction is only duplicated when it is applied to an argument in a named application. This is also connected to an optimization sometimes called *substituting abstractions on-demand*. For example, consider the term

$$t = w[w \leftarrow fx][f \leftarrow \lambda z_0.z_3[z_3 \leftarrow G(z_2)][z_2 \leftarrow G(z_1)][z_1 \leftarrow G(z_0)]][x \leftarrow \lambda y.s].$$

where $G(t) = \lambda w_0.w_3[w_3 \leftarrow w_1 w_2][w_2 \leftarrow gt][w_1 \leftarrow gt]$ with g a fixed name and s a normal term in positive λ-calculus. In the positive λ-calculus, after one `beta`-step and one `gc`-step, we obtain

$$z'_3[z'_3 \leftarrow G(z'_2)][z'_2 \leftarrow G(z'_1)][z'_1 \leftarrow G(x)][x \leftarrow \lambda y.s].$$

which is a normal term in the positive λ-calculus. However, in the VSC, we have

$$\begin{array}{ll}
z'_3[z'_3 \leftarrow G(z'_2)][z'_2 \leftarrow G(z'_1)][z'_1 \leftarrow G(x)][x \leftarrow \lambda y.s] & \to_{\mathtt{e}} \\
z'_3[z'_3 \leftarrow G(z'_2)][z'_2 \leftarrow G(z'_1)][z'_1 \leftarrow G(\lambda y.s)] & \to_{\mathtt{e}} \\
z'_3[z'_3 \leftarrow G(z'_2)][z'_2 \leftarrow G(G(\lambda y.s))] & \to_{\mathtt{e}} \\
z'_3[z'_3 \leftarrow G(G(G(\lambda y.s)))] & \to_{\mathtt{e}} \\
G(G(G(\lambda y.s))) &
\end{array}$$

which contains $2^3 = 8$ copies of $\lambda y.s$. From this example, we can see that the positive λ-calculus allows more sharing and avoids some possible exponential blow-ups that can occur in the VSC. Also, note that one has $\underline{s} = \underline{t}$ for $s \to_{\mathtt{e}} t$. As a result, these `e`-steps can be seen as redundant since they do not create any redex and can be ignored with the positive λ-calculus.

More formally, we can consider a variant of the VSC that treats substitutions in a linear style. That is, the `e`-rule is replaced by the following two rules:

$$\begin{array}{rll}
C\langle x\rangle[x \leftarrow L\langle v\rangle] \mapsto_{\mathtt{e}'} & L\langle C\langle v\rangle[x \leftarrow v]\rangle & \\
t[x \leftarrow L\langle v\rangle] \mapsto_{\mathtt{gc}'} & t & \text{if } x \notin fv(t)
\end{array}$$

Then the `beta`-rule can be expressed as a sequence of `m`, `e'`, and `gc'`-steps.

$$\begin{array}{r}
C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.L\langle y'\rangle] \to_{\mathtt{e}'} \\
C\langle t[z \leftarrow (\lambda y.L\langle y'\rangle)w]\rangle[x \leftarrow \lambda y.L\langle y'\rangle] \to_{\mathtt{m}} \\
C\langle t[z \leftarrow L\langle y'\rangle[y \leftarrow w]]\rangle[x \leftarrow \lambda y.L\langle y'\rangle] \to^*_{\mathtt{e}'}\to_{\mathtt{gc}'} \\
C\langle t[z \leftarrow L\langle y'\rangle\{y/w\}]\rangle[x \leftarrow \lambda y.L\langle y'\rangle] \to^*_{\mathtt{e}'}\to_{\mathtt{gc}'} \\
C\langle L\langle t\{z/y'\}\rangle\{y/w\}\rangle[x \leftarrow \lambda y.L\langle y'\rangle]
\end{array}$$

This shows that the reduction in the positive λ-calculus can be seen as a reduction strategy in the VSC that only applies *useful* substitutions to abstractions (a substitution is useful if it creates new redexes) and thus allows more sharing within a term. Therefore, we can say that the reduction of the positive λ-calculus works better than that of the VSC in terms of sharing on terms of the positive λ-calculus. However, we cannot say that the positive λ-calculus is better in general as the VSC contains more terms.

Furthermore, $\equiv_{\mathtt{str}}$ is a bisimulation with respect to $\to_{\mathtt{pos}}$.

Theorem 3. *Let t and u be two terms. If $t \equiv_{\mathtt{str}} u$ and $t \to_{\mathtt{pos}} t'$, then there exists u' such that $u \to_{\mathtt{pos}} u'$ and $t' \equiv_{\mathtt{str}} u'$.*

Proof. It suffices to see that $\sim_{\mathtt{str}}$ never creates or removes redexes.

$$L\langle \lambda x.t\rangle u \mapsto_{\mathtt{m}} L\langle t[x \leftarrow u]\rangle$$
$$t[x \leftarrow L\langle v\rangle] \mapsto_{\mathtt{e}} L\langle t\{x/v\}\rangle$$

Fig. 5. Reduction rules of the value substitution calculus [2]. Here, v is either a variable or an abstraction.

$$\begin{aligned} L\langle \lambda x.t\rangle u &\mapsto_{\mathtt{m}'} L\langle t[x \leftarrow u]\rangle \\ C\langle x\rangle[x \leftarrow u] &\mapsto_{\mathtt{e}'} C\langle u\rangle[x \leftarrow u] \\ t[x \leftarrow u] &\mapsto_{\mathtt{gc}'} t \qquad \text{if } x \notin fv(t) \end{aligned}$$

Fig. 6. Reduction rules of the linear substitution calculus [3].

A few more comments on the positive λ-calculus:

1. The positive bias syntax resembles the A-normal form [19]. A similar representation called *crumbled forms* has also been adapted in [5] to build an abstract machine for strong call-by-value. One issue common to calculi with explicit substitutions is the need for commutation rules to preserve specific syntactic forms. Terms in the positive bias syntax or crumbled forms can, however, be evaluated without using these rules.
2. While the `beta`-step keeps the sharing structure used to define the argument, there are still some redundancies in the positive λ-calculus. Consider the term $x_2[x_2 \leftarrow f x_1][x_1 \leftarrow f x_0][f \leftarrow \lambda x.z[z \leftarrow xy][y \leftarrow aa]]$. To eliminate the two `beta`-redexes $f x_0$ and $f x_1$, we have to make two copies of the abstraction body. Thus, the structure aa is introduced twice. To solve this redundancy, a possible solution is to *lift* the named structure $[y \leftarrow aa]$ to top-level before evaluation and obtain $x_2[x_2 \leftarrow f x_1][x_1 \leftarrow f x_0][f \leftarrow \lambda x.z[z \leftarrow xy]][y \leftarrow aa]$. This observation is often related to *full laziness* [29], a concept that has been widely studied in call-by-need settings, often using some graphical structures. However, we do not explore this aspect in this paper, and we leave it as a future work.

4 A graphical representation for terms: λ-graphs with bodies

In this section, we introduce a graphical representation for terms, called λ-graphs with bodies, that will be proved to capture the structural equivalence on Σ-terms given in Section 3. The definition of λ-graphs with bodies is split into two parts: we first define pre-graphs, and then define λ-graphs with bodies by giving additional structures and properties to deal with abstractions.

Definition 7. *A pre-graph is a directed acyclic graph built with the following three kinds of nodes:*

- *Application: an application node is labeled with* @ *and has two incoming edges (left and right). An application node is also called an* @*-node.*

- *Abstraction: an abstraction node is labeled with λ and has one incoming edge. Its only direct predecessor is called the output of the abstraction node. An abstraction node is also called a λ-node.*
- *Variable: a variable node has no incoming edge.*

A direct predecessor of a node is also called a child of the node.

Internal nodes (application and abstraction) of a pre-graph are used to represent intermediate expressions defined using constructs $[x \leftarrow p]$ within a term. We orient edges in such a way that there is an edge from n to m if and only if the definition of m requires the definition of n. In other words, nodes are defined in a *bottom-up* fashion.

In the following, we denote by $\mathcal{N}_\mathcal{G}$ and $\mathcal{E}_\mathcal{G} \subseteq \mathcal{N}_\mathcal{G} \times \mathcal{N}_\mathcal{G}$, respectively, the set of nodes and the set of edges of a graph $\mathcal{G}$.

Definition 8. *An unlabeled λ-graph with bodies is a pre-graph $\mathcal{G}$ together with two functions $bv : \Lambda_\mathcal{G} \to \mathcal{V}_\mathcal{G}$ and $body : \Lambda_\mathcal{G} \to 2^{\mathcal{N}_\mathcal{G} \setminus \mathcal{V}_\mathcal{G}}$ where $\Lambda_\mathcal{G}$ is the set of abstraction nodes of $\mathcal{G}$ and $\mathcal{V}_\mathcal{G}$ is the set of variable nodes of $\mathcal{G}$:*

1. *$body(l) \cap body(l') = \emptyset$ for $l \neq l'$.*
2. *The graph $\mathcal{B}_\mathcal{G} = (\Lambda_\mathcal{G}, \{(l, l') \mid l, l' \in \Lambda_\mathcal{G}, l \in body(l')\})$, called the scope graph of $\mathcal{G}$, is a DAG.*
3. *If a node n is $bv(l)$ or is in $body(l)$ and there is an edge $(n, m) \in \mathcal{E}_\mathcal{G}$, then we have*
   - *$m = l$, or*
   - *$m \in body(l')$ such that there is a path from l' to l in $\mathcal{B}_\mathcal{G}$. Note that this path is unique.*

We call $bv(l)$ the bound variable node and $body(l)$ the body of the abstraction node l. A node that does not belong to any body is called body-free and we denote by $body(\mathcal{G})$ the set of body-free non-variable nodes in $\mathcal{G}$. A free variable node is a variable node that is not a bound variable node and a global node is a body-free node that is not a bound variable node.

Intuitively, Point 3 of Definition 8 checks that every definition in a term is used in a valid scope: a name introduced in an abstraction can only be used within the abstraction.

Definition 9. *A well-labeled λ-graph with bodies, or simply a λ-graph with bodies, is an unlabeled λ-graph with bodies with a unique label assigned to each free variable node, and with a global node chosen, called the output of the λ-graph with bodies. A Σ-λ-graph with bodies is a λ-graph with bodies with a free variable node labeled by each element of a signature Σ.*

In order to visualize the maps $bv(\cdot)$ and $body(\cdot)$, we color the labels of abstraction nodes to distinguish them and color the frame of the nodes in their bodies with the same color. We proceed similarly for bound variables. In particular, a global node has its frame colored in black. Figure 7(a) shows a λ-graph with bodies, while Figure 7(b) shows an example that breaks Point 3 of Definition 8. In Figure 7(b), n belongs to the red body, m belongs to the blue body and there is an edge (n, m). m is not the red λ-node and there is no path from the blue λ-node to the red λ-node in the scope graph.

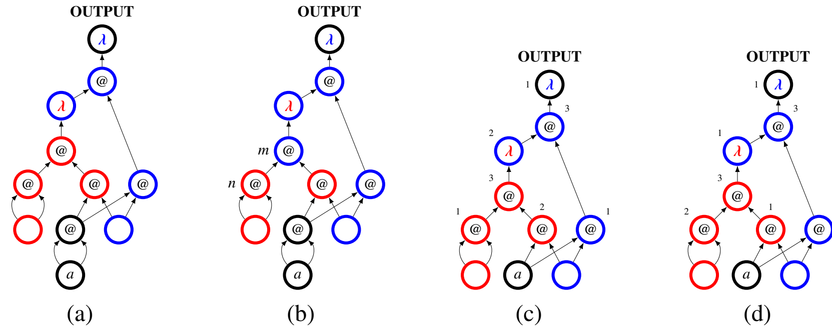

Fig. 7. Various figures.

5 Σ-λ-graphs with bodies and Σ-terms

In this section, we prove that there is a one-to-one correspondence between Σ-λ-graphs with bodies and Σ-terms up to $\equiv_{\texttt{str}}$. In order to establish such a correspondence, we first establish a one-to-one correspondence between ordered Σ-λ-graphs with bodies and Σ-terms where ordered Σ-λ-graphs with bodies are refinements of Σ-λ-graphs with bodies with some additional structure.

Dependency. Terms are expressed in a linear style. In other words, all the intermediate expressions within a term are defined in some (linear) order. However, graphs do not usually have this kind of structure. In order to establish the correspondence between terms and graphs, we have to give some more structure to our graphical representation. To do that, we first define dependency relations on nodes.

Definition 10. *Let $\mathcal{G}$ be a λ-graph with bodies and l an abstraction node. We define a relation $\prec_l$, called the dependency relation of l, on the set $body(l)$ of nodes as follows:*

- *$n \prec_l m$ if $(n, m) \in \mathcal{E}_\mathcal{G}$.*
- *$n \prec_l m$ if $(n, m') \in \mathcal{E}_\mathcal{G}$ for some $m' \in body(l')$ with $l' \neq l$, and there is a path from l' to m in $\mathcal{B}_\mathcal{G}$.*

Definition 11. *Let $\mathcal{G}$ be a λ-graph with bodies. We define the dependency relation $\prec_\mathcal{G}$ of $\mathcal{G}$ on the set $body(\mathcal{G})$ of body-free non-variable nodes of $\mathcal{G}$ as follows:*

- *$n \prec_\mathcal{G} m$ if $(n, m) \in \mathcal{E}_\mathcal{G}$.*
- *$n \prec_\mathcal{G} m$ if $(n, m') \in \mathcal{E}_\mathcal{G}$ for some $m' \in body(l)$, and there is a path from l to m in $\mathcal{B}_\mathcal{G}$.*

Definition 12. *Let $\mathcal{G}$ be a λ-graph with bodies. We define*

- *the dependency graph of $\mathcal{G}$, as the graph $\mathcal{D}_\mathcal{G} = (body(\mathcal{G}), \{(n, m) \mid n \prec_\mathcal{G} m\})$, and*
- *for all abstraction node l, the dependency graph of l, as the graph $\mathcal{D}_l = (body(l), \{(n, m) \mid n \prec_l m\})$.*

Proposition 3. *Let $\mathcal{G}$ be a λ-graph with bodies. Then we have:*

- *$\mathcal{D}_{\mathcal{G}}$ is a DAG, and*
- *for all abstraction node l of $\mathcal{G}$, $\mathcal{D}_l$ is a DAG.*

For example, for the λ-graph with bodies $\mathcal{G}$ in Figure 7(a), the dependency graph of the red (resp. blue) λ-node is the subgraph of $\mathcal{G}$ induced by its body, while the dependency graph $\mathcal{D}_{\mathcal{G}}$ of $\mathcal{G}$ has an edge from the application node to the blue λ-node.

As mentioned previously, our graphical representation should be equipped with some linear orderings on internal nodes. Moreover, these orderings should be compatible with the dependency relations defined above: they should be topological sorts of their corresponding dependency graphs.

Definition 13. *A topological sort of a directed graph $\mathcal{G}$ is a sequence containing each of its vertices such that for every edge (n, m), n appears before m in the sequence.*

Definition 14. *An ordered λ-graph with bodies $\hat{\mathcal{G}}$ is a λ-graph with bodies $\mathcal{G}$ together with a topological sort $T_{\mathcal{G}}$ of the graph $\mathcal{D}_{\mathcal{G}}$ and a topological sort T_l of the graph $\mathcal{D}_l$, for each l.*

Figures 7(c) and 7(d) show two ordered λ-graphs with bodies whose underlying λ-graphs with bodies are the same. As an example, the term corresponding to 7(c) is

$$x[x \leftarrow \lambda b_0.b_3[b_3 \leftarrow b_2 b_1][b_2 \leftarrow \lambda r_0.r_3[r_3 \leftarrow r_1 r_2][r_2 \leftarrow a b_0][r_1 \leftarrow r_0 r_0]][b_1 \leftarrow a b_0]].$$

Before giving a one-to-one correspondence between ordered λ-graphs with bodies and terms, we give a notion of *boxes* that is useful in the following.

Definition 15. *Let $\mathcal{G}$ be a λ-graph with bodies and l an abstraction node. We define the box of l as the union of bodies together with their bound variable nodes* below *l:*

$$box(l) = \bigcup_{l' \rightsquigarrow l \text{ in } \mathcal{B}_{\mathcal{G}}} (body(l') \cup \{bv(l')\})$$

where $l' \rightsquigarrow l$ in $\mathcal{B}_{\mathcal{G}}$ means that there is a path from l' to l in $\mathcal{B}_{\mathcal{G}}$.

In 7(a), the box of the red λ-node contains all the red-framed nodes while the box of the blue λ-node contains all the blue-framed and red-framed nodes.

Intuitively, for a λ-node l of a λ-graph with bodies $\mathcal{G}$, the graph obtained from the subgraph of $\mathcal{G}$ induced by $box(l)$ corresponds to the abstraction it introduces.

Ordered λ-graphs with bodies can actually be defined inductively as terms. We first give the following useful definitions.

Definition 16. *Let Σ be a signature and $x \in \Sigma$. We define $(x)_\Sigma$ as the ordered Σ-λ-graph with bodies that contains a free variable node labeled by each element of Σ and has the one labeled by x as the output.*

Definition 17. *Let Σ and Σ' be signatures, x, y, x_1, x_2 be names such that $\{x_1, x_2\} \subseteq \Sigma$, $x \notin \Sigma$ and $y \notin \Sigma'$, $\hat{\mathcal{G}}$ an ordered (Σ, x)-λ-graph with bodies and $\hat{\mathcal{G}}'$ an ordered (Σ', y)-λ-graph with bodies. Then*

- *$\hat{\mathcal{G}}\{\textbf{nd}\ x \leftarrow x_1@x_2\}$ is defined as the graph $\hat{\mathcal{H}}$ obtained from $\hat{\mathcal{G}}$ by replacing the free variable node labeled by x with an @-node whose left (resp. right) child is the variable node labeled by x_1 (resp. x_2). We then extend the topological sort $T_{\mathcal{G}}$ by having this application node as the minimal node. It is clear that $\hat{\mathcal{H}}$ is also an ordered Σ-λ-graph with bodies.*
- *$\hat{\mathcal{G}}\{\textbf{nd}\ x \leftarrow \lambda y.\hat{\mathcal{G}}'\}$ is defined as the graph $\hat{\mathcal{H}}$ obtained from by merging $\mathcal{G}$ and $\mathcal{G}'$ and by replacing the free variable node labeled by x with a new abstraction node l constructed as follows:*
  - *its only child is the output of $\mathcal{G}'$,*
  - *its bound variable is the free variable node labeled by y in $\mathcal{G}'$ (we erase the label y),*
  - *its body contains all the body-free non-variable nodes of $\mathcal{G}'$, and*
  - *its topological sort T_l is that of $\hat{\mathcal{G}}'$.*

  Note that $\mathcal{G}$ and $\mathcal{G}'$ can share some free variable nodes: they are merged so that there is only one free variable node labeled by each element of $\Sigma \cap \Sigma'$. In the end, we extend the topological sort $T_{\mathcal{G}}$ by having this new abstraction node as the minimal node. It is not difficult to see that $\hat{\mathcal{H}}$ is an ordered $(\Sigma \cup \Sigma')$-λ-graph with bodies.

Note that we can also use these definitions for λ-graphs with bodies by forgetting topological sorts.

Proposition 4. *Let Σ be a signature. Then $\hat{\mathcal{G}}$ is an ordered Σ-λ-graph with bodies if and only if $\Sigma \vdash \hat{\mathcal{G}}$ where $\Sigma \vdash \hat{\mathcal{G}}$ is defined by the following rules.*

$$x \in \Sigma\ \frac{}{\Sigma \vdash (x)_\Sigma}\ var \qquad \{y, z\} \subseteq \Sigma\ \frac{\Sigma, x \vdash \hat{\mathcal{G}}}{\Sigma \vdash \hat{\mathcal{G}}\{\textbf{nd}\ x \leftarrow y@z\}}\ @ \qquad \frac{\Sigma, y \vdash \hat{\mathcal{G}}' \quad \Sigma, x \vdash \hat{\mathcal{G}}}{\Sigma \vdash \hat{\mathcal{G}}\{\textbf{nd}\ x \leftarrow \lambda y.\hat{\mathcal{G}}'\}}\ \lambda$$

Proof. ($\Rightarrow$) Immediate from Definition 16 and Definition 17.

($\Leftarrow$) Let $\hat{\mathcal{G}} = (\mathcal{G}, T_{\mathcal{G}}, \{T_l \mid l \in \Lambda_{\mathcal{G}}\})$ be an ordered Σ-λ-graph with bodies. We proceed by induction on the number of non-variable nodes and consider each time the minimal node with respect to $T_{\mathcal{G}}$. Details can be found in Appendix B.

Note that the rules defining terms (*pvar*, *papp*, *pabs*) have the same structure as those in Proposition 4.

Theorem 4. *Let Σ be a signature. Then there is a one-on-one correspondence between ordered Σ-λ-graphs with bodies and Σ-terms.*

Proof. We can define translations $[\![\cdot]\!]_\Sigma$ from Σ-terms to ordered Σ-λ-graphs with bodies and $[\cdot]_\Sigma$ from ordered Σ-λ-graphs with bodies to Σ-terms by induction on the rules *pvar*, *papp*, *pabs* and those in Proposition 4. For the base cases, let $[\![x]\!]_\Sigma = (x)_\Sigma$ and $[(x)_\Sigma]_\Sigma = x$. We then have $[[\![t]\!]_\Sigma]_\Sigma = t$ and $[\![[\hat{\mathcal{G}}]_\Sigma]\!]_\Sigma = \hat{\mathcal{G}}$ for all Σ-term t and ordered Σ-λ-graph with bodies $\hat{\mathcal{G}}$.

We have established an isomorphism between Σ-terms and ordered Σ-λ-graphs with bodies. In Section 3, terms are considered equivalent up to $\equiv_{\mathtt{str}}$. How about ordered λ-graphs with bodies? It is natural to consider that ordered λ-graphs with bodies are equivalent if they share the same underlying λ-graph with bodies. The following proposition shows that topological sorts of a DAG can be connected to each other via *swaps*, similar to permutations of named structures for terms.

Proposition 5. *Let $\mathcal{G}$ be a DAG and S a topological sort of $\mathcal{G}$. We call swapping two non-adjacent nodes of $\mathcal{G}$ in a sequence of nodes a valid swap. Then a sequence of nodes can be obtained from S by a sequence of valid swaps if, and only if, it is a topological sort of $\mathcal{G}$.*

Proof. A proof is given in Appendix C.

The following theorem is a consequence of Theorem 4 and Proposition 5. Details can be found in Appendix D.

Theorem 5. *We have a one-to-one correspondence between Σ-λ-graphs with bodies and Σ-terms up to $\equiv_{\mathtt{str}}$.*

In the following, we also use $[\![\cdot]\!]_\Sigma$ (resp. $[\cdot]_\Sigma$) to denote the (bijective) map from the set of Σ-terms to the set of Σ-λ-graphs with bodies.

6 Substitution and reduction on λ-graphs with bodies

In this section, we show how substitution and reduction can be easily implemented on λ-graphs with bodies.

Substitution. Note that variable renaming, which is required in the case of terms, is not needed as internal nodes do not come with a label.

Definition 18 (Substitution on λ-graphs with bodies). *Let $\mathcal{G}$ be a Σ-λ-graph with bodies and $\mathcal{G}'$ a Σ'-λ-graph with bodies with $x \notin \Sigma'$. We define the substitution of x for $\mathcal{G}'$ in $\mathcal{G}$, written $\mathcal{G}[x/\mathcal{G}']$, as the $((\Sigma \setminus \{x\}) \cup \Sigma')$-$\lambda$-graph with bodies obtained from by merging $\mathcal{G}'$ into $\mathcal{G}$ and, if $x \in \Sigma$ by replacing the free variable node labeled by x with the output node of $\mathcal{G}'$. Note that we have to merge common free variable nodes labeled by elements of $(\Sigma \setminus \{x\}) \cap \Sigma'$ and the output node of $\mathcal{G}[x/\mathcal{G}']$ is that of $\mathcal{G}$.*

In Figure 8, we present an example for the substitution on λ-graphs with bodies. From this example, we can clearly see that the structure of bodies is kept under substitution.

The substitution on λ-graphs with bodies implements indeed the substitution on terms.

Theorem 6. *Let t be a Σ-term and u a Σ'-term such that $x \notin \Sigma'$. Then $[\![t[x/u]]\!]_{(\Sigma\setminus\{x\})\cup\Sigma'} = [\![t]\!]_\Sigma[x/[\![u]\!]_{\Sigma'}]$.*

Proof. A straightforward induction on u.

Reduction. We now show how to implement the two rewrite rules on λ-graphs with bodies. We first define *contexts* for λ-graphs with bodies using the following grammar:

$$C \coloneqq \square \mid C[\mathbf{nd}\ x \leftarrow y@z] \mid C[\mathbf{nd}\ x \leftarrow \lambda y.\mathcal{G}] \mid \mathcal{G}[\mathbf{nd}\ x \leftarrow \lambda y.C]$$

The *plugging* $C\langle\mathcal{G}\rangle$ of a λ-graph with bodies $\mathcal{G}$ in the context C is defined inductively by:

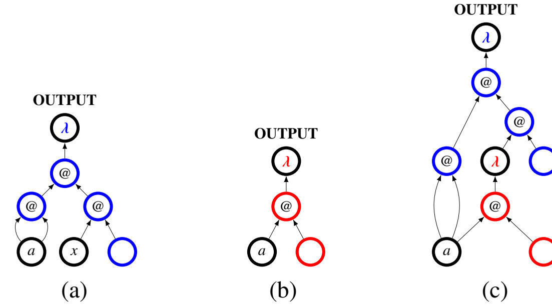

Fig. 8. An example for the substitution on λ-graphs with bodies: (c) is the result of substituting the free variable node x for (b) in (a).

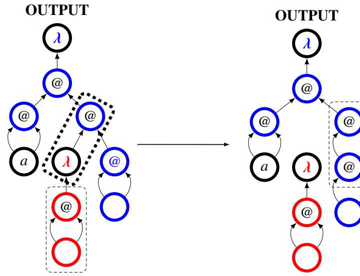

Fig. 9. An example of `beta`-reduction on λ-graphs with bodies. The thick dashed box is a `beta`-redex and the two thin dashed boxes in the λ-graphs with bodies correspond to the box of the red λ-node and its copy, respectively.

$$\begin{aligned} \Box\langle \mathcal{G} \rangle &= \mathcal{G} \\ C[\mathbf{nd}\ x \leftarrow y@z]\langle \mathcal{G} \rangle &= C\langle \mathcal{G} \rangle\{\mathbf{nd}\ x \leftarrow y@z\} \\ C[\mathbf{nd}\ x \leftarrow \lambda y.\mathcal{G}']\langle \mathcal{G} \rangle &= C\langle \mathcal{G} \rangle\{\mathbf{nd}\ x \leftarrow \lambda y.\mathcal{G}'\} \\ \mathcal{G}'[\mathbf{nd}\ x \leftarrow \lambda y.C]\langle \mathcal{G} \rangle &= \mathcal{G}'\{\mathbf{nd}\ x \leftarrow \lambda y.C\langle \mathcal{G} \rangle\}. \end{aligned}$$

The `gc`-rule can be defined by erasing an abstraction node with no parent and its box. A `beta`-redex is simply an @-node that has a λ-node as its left child. To eliminate it, it suffices to duplicate the *box* of the λ-node, replace the @-node with the copy of the output node of λ-node, and then replace the bound variable in this copy with the argument, i.e., the right child of the @-node. One should be careful about the structure of bodies: in this copy, all the nodes that were in the body of the λ-node should be moved to the same body as the argument or to the corresponding body if the argument is a bound variable. Figure 9 shows a `beta`-reduction step on λ-graphs with bodies. We denote by $\to_{\mathtt{G}}$ the contextual closure of these two steps (`gc` and `beta`).

The translation $[\![\cdot]\!]_\Sigma$ is a strong bisimulation between the positive λ-calculus and λ-graphs with bodies with $\to_{\mathtt{G}}$.

Theorem 7. *Let s and t be two Σ-terms. Then $s \to_{\mathtt{pos}} t$ if and only if $[\![s]\!]_\Sigma \to_{\mathtt{G}} [\![t]\!]_\Sigma$.*

Proof. This is a consequence of Theorem 6.

In general, strong bisimulations preserve confluence quotient by the translation. Here, the quotient induced by the translation $[\![\cdot]\!]_\Sigma$ on Σ-λ-graphs with bodies is the identity, so the confluence coincides with the confluence modulo quotient by $[\![\cdot]\!]_\Sigma$. Hence, we have the confluence of $\to_{\mathtt{G}}$ from that of $\to_{\mathtt{pos}}$.

Theorem 8. $\to_{\mathtt{G}}$ *is confluent.*

Proof. Straightforward from Theorem 2, Theorem 3 and Theorem 7.

Relations with other graphical representations. In [2], Accattoli revealed a close relationship between the proof nets and the value substitution calculus. As mentioned earlier, the VSC is equipped with a small-step but not a micro-step $\mathtt{e}$-rule. By replacing the $\mathtt{e}$-rule by its micro-step variant, one is actually able to simulate the positive λ-calculus in the VSC. This is also true on the graphical side. However, we have a slightly different treatment of free variables here: free variable nodes are considered shared between all the bodies, so no free variable node is explicitly included in any body. This approach is suggested by the rules *papp* and *pabs* where the left hand side of the conclusion is entirely included in those of the premises (this is actually a feature of *LJF*, where weakenings are delayed and only allowed in initial rules I_l and I_r). Due to this choice, the content of a box is no longer a λ-graph with bodies (one has to include all the free variable nodes), which makes induction arguments a bit more complicated. However, it provides us a clear way to *sequentialize* λ-graphs with bodies.

In [14], λ-graphs are used to study the sharing equality. As the name suggests, λ-graphs with bodies can be (almost) seen as an extension of λ-graphs with bodies. They only differ in the following points:

- λ-graphs are not pointed: there is no unique output assigned to a λ-graph as their goal is to study if two terms have the same encoding *under the same context*.
- There is no *useless* node under an abstraction in λ-graphs: an abstraction node has a unique child (output). Without the notion of bodies, there is no way to define nodes that are under an abstraction but not used to define the output. This is not a drawback as these useless nodes do not affect the unfoldings of terms.

7 Generalization

In this section, we explain briefly how this graphical representation can be generalized using *LJF* and different polarized theories.

In Definition 3, *LJ* can be extended by any polarized theory of order one or two. In our study of untyped λ-terms, we use exactly one formula of order one ($D \supset D \supset D$) and one formula of order two ($(D \supset D) \supset D$). Our graphical representation can be generalized to any *positively polarized* theory of order one or two in the following way.

Each node comes with a *type* which is an atomic formula. A formula of order one or two can be written as $B_1 \supset \cdots \supset B_n \supset A$ with $n \geq 1$, B_i of order at most one and A atomic. This formula corresponds to a node *of type* A that has n incoming edges, each

of which corresponds to one B_i. For $1 \leq i \leq n$, if B_i is atomic, then it corresponds to simply a node of type B_i, and if B_i is of order one, then it comes with a notion of *body*.

A similar notion of reduction can also be defined: a redex is simply a group of nodes that follow a certain pattern, and reduction rules can be defined using the children (or boxes) of these nodes.

8 Conclusion

We propose the positive λ-calculus based on the encoding of untyped λ-terms defined using the positive bias assignment in [27]. This calculus features a call-by-value rewrite system and is closely related to the value substitution calculus.

We introduce a graphical representation for terms of the positive λ-calculus, called λ-graphs with bodies and show how operations such as substitution and reduction can be implemented on this structure.

Using the focused proof system *LJF* as a framework to build term structures makes it possible to generalize the results in this paper to other kinds of term calculi.

Future work. We plan to explore at least the following directions in the future:

- Fully lazy sharing: As mentioned, the reduction procedure of the positive λ-calculus is not perfect since all the named structures within an abstraction are duplicated. We hope to explore the possibilities of allowing more sharing along the reduction and works on full laziness can surely provide more insights.
- Mixing positive and negative term structures: An important feature of *LJF* is *polarization*. This paper focuses on the case that atoms are all given the positive polarity. What if we consider atoms of different polarities? Will this allow expressing more term structures while having a good notion of sharing? These are the questions we hope to answer in our follow-up study.
- Connections with the VSC: The VSC has been applied to study the call-by-value λ-calculus and various topics related to it: abstract machines, sharing, etc. Also, a correspondence between VSC-terms and proof nets has been established. In this paper, we show how these two calculi are similar but different at the same time. It seems natural and interesting to look for more connections between these two calculi and to see if the positive λ-calculus can provide a different perspective on the topics mentioned above.

Acknowledgement. I would like to thank Dale Miller and Beniamino Accattoli for their valuable discussions and suggestions. I am also grateful to the anonymous reviewers for their helpful comments.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. Journal of Functional Programming **1**(4), 375–416 (Oct 1991)
2. Accattoli, B.: Proof nets and the call-by-value λ-calculus. Journal of Theoretical Computer Science (TCS) (2015). https://doi.org/10.1016/j.tcs.2015.08.006

3. Accattoli, B.: Proof nets and the linear substitution calculus. In: International Colloquium on Theoretical Aspects of Computing. pp. 37–61. Springer (2018)
4. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 659–670 (2014)
5. Accattoli, B., Condoluci, A., Coen, C.S.: Strong call-by-value is reasonable, implosively. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–14. IEEE (2021)
6. Accattoli, B., Paolini, L.: Call-by-value solvability, revisited. In: Functional and Logic Programming: 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings 11. pp. 4–16. Springer (2012)
7. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation **2**(3), 297–347 (1992). https://doi.org/10.1093/logcom/2.3.297
8. Andreoli, J.M.: Focussing and proof construction. Annals of Pure and Applied Logic **107**(1), 131–163 (2001)
9. Balabonski, T.: A unified approach to fully lazy sharing. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 469–480 (2012)
10. Brock-Nannestad, T., Guenot, N., Gustafsson, D.: Computation in focused intuitionistic logic. In: Falaschi, M., Albert, E. (eds.) Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14–16, 2015. pp. 43–54. ACM (2015). https://doi.org/10.1145/2790449.2790528
11. Chaudhuri, K.: Focusing strategies in the sequent calculus of synthetic connectives. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR: International Conference on Logic, Programming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 467–481. Springer (Nov 2008). https://doi.org/10.1007/978-3-540-89439-1_33
12. Chaudhuri, K., Hetzl, S., Miller, D.: A multi-focused proof system isomorphic to expansion proofs. Journal of Logic and Computation **26**(2), 577–603 (2016)
13. Chaudhuri, K., Miller, D., Saurin, A.: Canonical sequent proofs via multi-focusing. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Fifth International Conference on Theoretical Computer Science. IFIP, vol. 273, pp. 383–396. Springer (Sep 2008). https://doi.org/10.1007/978-0-387-09680-3_26
14. Condoluci, A., Accattoli, B., Coen, C.S.: Sharing equality is linear. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming. pp. 1–14 (2019). https://doi.org/10.1145/3354166.3354174
15. Danos, V., Joinet, J.B., Schellinx, H.: LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic, pp. 211–224. No. 222 in London Mathematical Society Lecture Note Series, Cambridge University Press (1995). https://doi.org/10.1017/CBO9780511629150
16. Delande, O., Miller, D.: A neutral approach to proof and refutation in MALL. In: Pfenning, F. (ed.) 23th Symp. on Logic in Computer Science. pp. 498–508. IEEE Computer Society Press (2008). https://doi.org/10.1016/j.apal.2009.07.017
17. Di Cosmo, R., Kesner, D.: Strong normalization of explicit substitutions via cut elimination in proof nets. In: Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science. pp. 35–46 (1997). https://doi.org/10.1109/LICS.1997.614927
18. Dyckhoff, R., Lengrand, S.: LJQ: a strongly focused calculus for intuitionistic logic. In: Beckmann, A., *et al.* (eds.) Computability in Europe 2006. LNCS, vol. 3988, pp. 173–185. Springer (2006). https://doi.org/10.1007/11780342_19

19. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation. pp. 237–247 (1993)
20. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
21. Herbelin, H.: A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In: Computer Science Logic, 8th International Workshop, CSL '94. LNCS, vol. 933, pp. 61–75. Springer (1995). https://doi.org/10.1007/BFb0022247
22. Hughes, D.J.D.: Proofs without syntax. Annals of Mathematics **143**(3), 1065–1076 (Nov 2006)
23. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theoretical Computer Science **410**(46), 4747–4768 (2009). https://doi.org/10.1016/j.tcs.2009.07.041, abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi
24. Marin, S., Miller, D., Pimentel, E., Volpe, M.: From axioms to synthetic inference rules via focusing. Annals of Pure and Applied Logic **173**(5), 1–32 (2022). https://doi.org/10.1016/j.apal.2022.103091
25. Miller, D.: A compact representation of proofs. Studia Logica **46**(4), 347–370 (1987). https://doi.org/10.1007/BF00370646
26. Miller, D., Saurin, A.: From proofs to focused proofs: a modular proof of focalization in linear logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007: Computer Science Logic. LNCS, vol. 4646, pp. 405–419. Springer (2007)
27. Miller, D., Wu, J.H.: A positive perspective on term representations. In: Klin, B., Pimentel, E. (eds.) 31st EACSL Annual Conference on Computer Science Logic (CSL 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 252, pp. 3:1–3:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.CSL.2023.3
28. Pimentel, E., Nigam, V., Neto, J.: Multi-focused proofs with different polarity assignments. In: Benevides, M., Thiemann, R. (eds.) Proc. of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015). ENTCS, vol. 323, pp. 163–179 (Jul 2016). https://doi.org/10.1016/j.entcs.2016.06.011
29. Wadsworth, C.P.: Semantics and Pragmatics of the Lambda Calculus. Ph.D. thesis, University of Oxford (1971)

A Proof of Proposition 1

To prove this, we define the unfolding $\underline{C}$ of a context C inductively:

$$
\begin{aligned}
\underline{\Box} &= \diamond \\
\underline{C[x \leftarrow yz]} &= \underline{C}\{x/yz\} \\
\underline{C[x \leftarrow \lambda y.s]} &= \underline{C}\{x/\lambda y.\underline{s}\} \\
\underline{s[x \leftarrow \lambda y.C]} &= \underline{s}\{x/\lambda y.\underline{C}\}
\end{aligned}
$$

Here $\diamond$ is a variable and $\underline{C}$ is an untyped λ-term. We have $\underline{C\langle t\rangle} = \underline{C}\{\diamond/\underline{t}\}$ by a straightforward induction. The case of $\rightarrow_{\mathsf{gc}}$ is trivial. It suffices now to prove

$$
\underline{C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.s]} \rightarrow^{*}_{\mathsf{pos}} \underline{C\langle t\{z/s\{y/w\}\}\rangle[x \leftarrow \lambda y.s]}.
$$

We have

$$
\begin{aligned}
&\underline{C\langle t[z \leftarrow xw]\rangle[x \leftarrow \lambda y.s]} \\
&= \underline{C\langle t[z \leftarrow xw]\rangle}\{x/\lambda y.\underline{s}\} \\
&= \underline{C}\{\diamond/\underline{t[z \leftarrow xw]}\}\{x/\lambda y.\underline{s}\} \\
&= \underline{C}\{\diamond/\underline{t}\{z/xw\}\}\{x/\lambda y.\underline{s}\}
\end{aligned}
$$

and

$$
\begin{aligned}
&\underline{C\langle t\{z/s\{y/w\}\}\rangle[x \leftarrow \lambda y.s]} \\
&= \underline{C\langle t\{z/s\{y/w\}\}\rangle}\{x/\lambda y.\underline{s}\} \\
&= \underline{C}\{\diamond/\underline{t\{z/s\{y/w\}\}}\}\{x/\lambda y.\underline{s}\} \\
&= \underline{C}\{\diamond/\underline{t}\{z/\underline{s}\{y/w\}\}\}\{x/\lambda y.\underline{s}\}.
\end{aligned}
$$

It is clear that

$$
\underline{C}\{\diamond/\underline{t}\{z/xw\}\}\{x/\lambda y.\underline{s}\} \rightarrow^{*}_{\mathsf{pos}} \underline{C}\{\diamond/\underline{t}\{z/\underline{s}\{y/w\}\}\}\{x/\lambda y.\underline{s}\}.
$$

B Proof of Proposition 4

($\Rightarrow$) Immediate from Definition 16 and Definition 17.

($\Leftarrow$) Let $\hat{\mathcal{G}} = (\mathcal{G}, T_{\mathcal{G}}, \{T_l \mid l \in \Lambda_{\mathcal{G}}\})$ be an ordered Σ-λ-graph with bodies. We proceed by induction on the number of non-variable nodes in $\mathcal{G}$.

- $\mathcal{G}$ contains only variable nodes and its output is labeled by x: $\hat{\mathcal{G}} = (x)_{\Sigma}$.
- $\mathcal{G}$ contains a non-variable node: in this case, $\mathcal{G}$ contains at least one body-free non-variable node. We consider the minimal one n with respect to $T_{\mathcal{G}}$.

- application: both of its children are free variable nodes (otherwise, it would not have been the minimal body-free non-variable node), labeled by y and z, respectively. Let $\hat{\mathcal{G}}'$ be the graph obtained from $\hat{\mathcal{G}}$ by replacing this application node with a fresh free variable node x (with topological sorts inherited). This graph $\hat{\mathcal{G}}'$ is an ordered (Σ, x)-λ-graph with bodies and we have $\hat{\mathcal{G}} = \hat{\mathcal{G}}'\{\mathbf{nd}\ x \leftarrow y@z\}$.
- abstraction: Consider the subgraph $\mathcal{H}$ of $\mathcal{G}$ induced by $box(n) \cup \Sigma$. By giving the bound variable node of n in $\mathcal{G}$ a label y, we obtain an ordered (Σ, y)-λ-graph with bodies $\hat{\mathcal{H}}'$ (with topological sorts inherited). To prove $\mathcal{H}'$ is indeed a λ-graph with bodies, it suffices to prove that "for every node m in $box(n)$, every child of m in $\mathcal{G}$ is either a free variable node or is also in $box(n)$". Let m be a node in $box(n)$ and m' be a child of m that is not in $box(n)$. We distinguish two cases:
  - $*$ m' is body-free and is not a bound variable node: then it is a free-variable node. Otherwise, n would not have been minimal w.r.t. $T_{\mathcal{G}}$ since we have $m' \prec_{\mathcal{G}} n$ in this case, a contradiction.
  - $*$ m' belongs to $body(l)$ or is $bv(l)$ for some abstraction node l: by Definition 15, m is in $body(n')$ (m cannot be a bound variable as it has a child) for some n' such that there is a path $n' \rightsquigarrow n$ in $\mathcal{B}_{\mathcal{G}}$. Then by Condition 3 of Definition 8, we have a path $n' \rightsquigarrow l$ in $\mathcal{B}_{\mathcal{G}}$. Since n is body-free and by the uniqueness of paths between two nodes in $\mathcal{B}_{\mathcal{G}}$ (by Conditions 1 and 2 of Definition 8), we have a path from l to n in $\mathcal{B}_{\mathcal{G}}$, which implies that m' is in $box(n)$ by Definition 15, a contradiction.

  Now by removing all the nodes in $box(n)$ from $\mathcal{G}$ and by replacing n with a fresh free variable node x, we get an ordered (Σ, x)-λ-graph with bodies $\hat{\mathcal{G}}'$. Similarly, to prove that $\mathcal{G}'$ is a λ-graph with bodies, it suffices to check that the children of all its nodes except n in $\mathcal{G}$ are still in $\mathcal{G}'$, i.e., not in $box(n)$. Suppose that there is a node m different from n in $\mathcal{G}'$ having a child m' (in $\mathcal{G}$) in $box(n)$. Assume that m' is $bv(n')$ or in $body(n')$. Then by Condition 3 of Definition 8, we have either (1) $m = n'$ or (2) $m \in body(l)$ such that there is a path $l \rightsquigarrow n'$ in $\mathcal{B}_{\mathcal{G}}$. The first case implies that m' is $bv(m)$ or in $body(m)$ and since m' is in $box(n)$, we have a path $m \rightsquigarrow n$ in $\mathcal{B}_{\mathcal{G}}$. The path is non-empty (m is different from n), so m is also in $box(n)$, a contradiction. For the second case, since m' is in $box(n)$, we have a path $n' \rightsquigarrow n$ in $\mathcal{B}_{\mathcal{G}}$ by Definition 15. Thus we have a path $l \rightsquigarrow n$ in $\mathcal{B}_{\mathcal{G}}$, which is impossible since m is in $body(l)$ and m is not in $box(n)$. Then we have $\hat{\mathcal{G}} = \hat{\mathcal{G}}'\{\mathbf{nd}\ x \leftarrow \lambda y.\hat{\mathcal{H}}'\}$.

C Proof of Proposition 5

We only prove the converse implication here since the direct implication is trivial. Assume that $S = S_1, \ldots, S_k$. Let $T = T_1, \ldots, T_k$ be a topological sort of $\mathcal{G}$. Suppose that i is the minimal integer such that $S_i \neq T_i$. We have thus $S_1 = T_1, \ldots, S_{i-1} = T_{i-1}$. Now we construct a sequence S' from applying a sequence of valid swaps to S such that

$$S'_1 = T_1, \ldots, S'_i = T_i. \tag{1}$$

Let j be the unique integer such that $S_j = T_i$. It is clear that $j > i$. We now swap S_j with S_{j-1}. This is a valid swap since S is a topological sort and the sequence obtained is still a topological sort. By repeating this step, we can obtain a topological sort S' by moving S_j to the i-th position and S' satisfies clearly (1).

Now repeat the step by considering S' instead of S, and so on. We can eventually reach T by a sequence of valid swaps.

D Proof of Theorem 5

We first define bodies and dependency relations on terms in a similar way.

Definition 19. *Let t be a Σ-term. Let x be a name introducing an abstraction in t, i.e., we have the pattern $[x \leftarrow \lambda y.s]$ in t. We define the body of x, written $body(x)$, as the set $\{x_1, \cdots, x_k\}$ of names where s is of the form $x_{k+1}[x_k \leftarrow p_k] \cdots [x_1 \leftarrow p_1]$. We say that y is the bound variable of x, denoted by $bv(x)$. It is clear that any name introduced by the construct $[x \leftarrow p]$ belongs to at most one body. We denote by $body(t)$ the set of names introduced by some construct $[x \leftarrow p]$ that do not belong to any body of any name introducing an abstraction.*

Definition 20. *Let t be a Σ-term. Let x be a name introduced by some $[x \leftarrow p]$ in t. We define the dependency set of x, written $dep(x)$, as the set $fv(p)$. We then define the dependency graphs of t and of its names introducing abstractions:*

- *The dependency graph $\mathcal{D}_t$ of t is defined as the graph*

$$(body(t), \{(x, y) \mid x, y \in body(t) \text{ and } x \in dep(y)\}).$$

- *The dependency graph $\mathcal{D}_x$ of a name x introducing an abstraction is defined as the graph*

$$(body(x), \{(y, z) \mid y, z \in body(x) \text{ and } y \in dep(z)\}).$$

Note that if $x \in dep(y)$, then x cannot be defined later than y in the same body. This observation leads to the following proposition.

Proposition 6. *Let t be a Σ-term. Then we have:*

- *$\mathcal{D}_t$ is a DAG.*
- *$\mathcal{D}_x$ is a DAG for any x introducing an abstraction.*

The permutations defining structural equivalence are applied to two consecutive named structures in the same body and we can permute them if and only if neither of them is in the dependency set of the other. Thus, we have the following proposition.

Proposition 7. *Two consecutive named structures can be permuted using $\sim_{\mathtt{str}}$ if and only if they are not adjacent in their corresponding dependency graph.*

The following key lemma can be proved by induction on terms.

Lemma 1. *For any Σ-term t, t and $[\![t]\!]_\Sigma$ have the same dependency graphs.*

Theorem 5 is simply a consequence of Theorem 4, Proposition 5, Proposition 7, and Lemma 1.