A Positive Perspective on Term Representation

Dale Miller
Inria Saclay, Palaiseau, France
LIX, Institut Polytechnique de Paris, France

Jui-Hsuan Wu
LIX, Institut Polytechnique de Paris, France

Abstract

We use the focused proof system *LJF* as a framework for describing term structures and substitution. Since the proof theory of *LJF* does not pick a canonical polarization for primitive types, two different approaches to term representation arise. When primitive types are given the negative polarity, *LJF* proofs encode terms as tree-like structures in a familiar fashion. In this situation, cut elimination also yields the familiar notion of substitution. On the other hand, when primitive types are given the positive polarity, *LJF* proofs yield a structure in which explicit sharing of term structures is possible. Such a representation of terms provides an explicit method for sharing term structures. In this setting, cut elimination yields a different notion of substitution. We illustrate these two approaches to term representation by applying them to the encoding of untyped λ-terms. We also exploit concurrency theory techniques – namely traces and simulation – to compare untyped λ-terms using such different structuring disciplines.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases term representation, sharing, focused proof systems

Digital Object Identifier 10.4230/LIPIcs.CSL.2023.3

Category Invited Talk

Related Version *Full Version*: `https://hal.inria.fr/hal-03843587`

Acknowledgements We thank Beniamino Accattoli and Kaustuv Chaudhuri for their valuable discussions and suggestions. We also thank anonymous reviewers for their comments on an earlier draft of this paper.

1 Introduction

The structure of terms and expressions are represented variously via labeled trees or directed acyclic graphs (DAGs). When such expressions contain bindings, additional devices are needed. We follow a familiar line of research in which the design of term representations is motivated by proof-theoretic considerations. Accordingly, we rely on proof theory in the hope that it will provide careful and formal descriptions of the term structures that serve as the foundation of theorem provers, semantic specifications, interpreters, parsers, and compilers.

Applications of structural proof theory usually start with either natural deduction or sequent calculus proof systems, both of which were introduced by Gentzen in [15]. For our purposes here, these two proof systems are inadequate: natural deduction (with or without *generalized elimination rules* [36]) seems too restrictive, and sequent calculus seems too unstructured. Instead, we use the *LJF* proof system [24], which results from applying the notions of *polarity* and *focusing* [1, 17] to Gentzen's *LJ* proof system as a framework for studying term structures with and without bindings and with and without explicit sharing constructs. By examining the dynamics of cut-elimination in *LJF*, we can also describe substitution into such term structures.

Focused proof theories have been successfully applied within the Curry-Howard correspondence: for example, call-by-value, call-by-name, and call-by-push-value evaluation strategies have been related to different choices in polarizing type expressions [34, 39, 42]. Instead of dealing with functional programming style evaluation, we address a more primitive notion: what is a *term* (used to encode programs), and what is *substitution* (used to describe various kinds of evaluation). Even this more narrow setting has been addressed using the focused proof systems *LJT* [12, 22] and *LJQ* [11]. As in those papers, we shall consider proofs-as-terms and not proofs-as-programs since we do not consider the evaluation of such structures in this paper.

2 The LJF proof system

Gentzen's sequent calculus proof system *LJ* [15] employs rather tiny and slippery inference rules: they are tiny since most of them deal with at most one logical connective at a time, and they are slippery since the order in which they are applied can often be freely reorganized. As a result, it is hard to use *LJ* to identify large-scale structures in proofs. For example, capturing the two phases of proof search in logic programming – *goal-reduction* and *backchaining* – with sequent calculus rules requires rather technical arguments about permutations of inference rules [30]. Andreoli's invention of a *focused proof system* for linear logic [1] provided means for adding more structure to sequent calculus proofs for linear logic. This notion of focusing was eventually moved to intuitionistic and classical logic as the *LJF* and *LKF* proof systems [24]. This paper considers only intuitionistic logic and only for formulas built using implication as the only logical connective.

2.1 LJF inference rules

There are two kinds of sequents in *LJF*: the $\Uparrow$-sequents $\Gamma \Uparrow \Theta \vdash \Delta \Uparrow \Delta'$ and the $\Downarrow$-sequents $\Gamma \Downarrow \Theta \vdash \Delta \Downarrow \Delta'$. Here, all four *zones* Γ, Θ, Δ, and Δ' are multisets of formulas. Given that we are in an intuitionistic proof system, we require that the multiset union $\Delta \cup \Delta'$ is always a singleton. The zones Γ and Δ' are called the left and right *storage zones*. The zones Θ and Δ are called the left and right *staging areas*. As these names suggest, formulas in the storage zones can persist during the construction of a proof, while formulas in the staging area are intended to have a limited impact on the long-term construction of such a proof. Occurrences of logical connectives introduced by the left and right introduction rules only appear in the staging area. As we shall see, the only sequents with empty staging areas are of the form $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow \Delta$. Such sequents are called *border sequents*: when we define synthetic inference rules in Definition 6, these sequents form the borders (the conclusion and premises) of synthetic inference rules.

Notational conventions: We usually denote an empty zone by explicitly using the dot $\cdot$. Also, while every *LJF* sequent has two occurrences of either $\Uparrow$ or $\Downarrow$, we write fewer of these arrows by adopting the convention that we drop writing $\cdot \Downarrow$ and $\cdot \Uparrow$ when they appear on the right, and we drop writing $\Downarrow \cdot$ and $\Uparrow \cdot$ when they appear on the left. Finally, since the right side of sequents have exactly one formula, we replace writing $\Downarrow \cdot$ with $\Downarrow$ and $\Uparrow \cdot$ with $\Uparrow$. Thus, the sequent $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow E$ can be written as $\Gamma \vdash E$ and the sequent $\Gamma \Downarrow \cdot \vdash E \Downarrow \cdot$ can be written as $\Gamma \vdash E \Downarrow$. As a result of these conventions, border sequents in *LJF* will resemble sequents in *LJ*, which is a completely desirable resemblance.

In the general setting, Gentzen's *LJ* proof system involves *unpolarized* formulas, while the *LJF* focused proof system involves *polarized formulas*. Since, in this paper, we are only interested in one logical connective, the implication $\supset$, and since the polarization of an

Decide, Release, and Store Rules

$$\frac{N, \Gamma \Downarrow N \vdash A}{N, \Gamma \vdash A}\, D_l \qquad \frac{\Gamma \vdash P \Downarrow}{\Gamma \vdash P}\, D_r \qquad \frac{\Gamma \Uparrow P \vdash A}{\Gamma \Downarrow P \vdash A}\, R_l \qquad \frac{\Gamma \vdash N \Uparrow}{\Gamma \vdash N \Downarrow}\, R_r \qquad \frac{\Gamma, C \Uparrow \Theta \vdash \Delta' \Uparrow \Delta}{\Gamma \Uparrow \Theta, C \vdash \Delta' \Uparrow \Delta}\, S_l \qquad \frac{\Gamma \Uparrow \Theta \vdash A}{\Gamma \Uparrow \Theta \vdash A \Uparrow}\, S_r$$

Initial Rules

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

Introduction Rules for implication

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

Figure 1 The rules of (cut-free) *LJF* for the implicational fragment of propositional intuitionistic logic.

implication is unambiguous (it is *negative*), the only distinction between unpolarized and polarized formulas falls on atomic formulas: in polarized formulas, we need to describe the polarity of atomic formulas explicitly. (We remind the reader that a logical connective is polarized negatively if its right introduction rule is invertible.)

▶ **Definition 1.** *An* atomic bias assignment *is a function, δ, that maps atomic formulas to either $+$ or $-$. A formula B is* negative *if it is either an implication or atomic and $\delta(B) = -$. A formula B is* positive *if it is atomic and $\delta(B) = +$.*

The *LJF* proof system for intuitionistic propositional logic over just implication is given in Figure 1. This figure uses the following schema variables: P is a positive formula, N is a negative formula, A is an atomic formula, and B, B', and C denote arbitrary formulas. Note that in our simplified setting, a positive formula is also atomic.

An $\Uparrow$-*phase* is a collection of occurrences of $\Uparrow$-sequents that are all connected via inference rules; similarly, a $\Downarrow$-*phase* is a collection of occurrences of $\Downarrow$-sequents that are all connected via inference rules. The *decide* rules D_l and D_r are the only inference rules (in a cut-free proof) that have a border sequent as a conclusion. Thus, the decide rules sit on top of an $\Uparrow$-phase while their premises are at the bottom of a $\Downarrow$-phase. The *store* rules S_l and S_r work within an $\Uparrow$-phase. The release rules R_l and R_r are dual to the decide rules in that they sit on top of a $\Downarrow$-phase while their premises are at the bottom of an $\Uparrow$-phase.

When searching for a proof in *LJF*, the only occurrence of *don't know nondeterminism* occurs with the decide rules. There is a possible choice to decide on the right or left (D_l or D_r), and if D_l is selected, then another important choice is which formula in the left storage should be selected.

The following soundness and completeness theorem for *LJF* can be found in [24], where these theorems are proved for full first-order intuitionistic logic. (That paper also proves that *LJF* captures the structure of *LJT* and *LJQ* as well as some hybrid forms of those two proof systems.)

▶ **Theorem 2.** *Let B be an unpolarized formula composed only of implications and atomic formulas.*

1. *If B is provable in LJ and if $\delta(\cdot)$ is any atomic bias assignment for the atoms in B, then the sequent $\cdot \Uparrow \cdot \vdash B \Uparrow \cdot$ is provable in LJF.*
2. *If $\delta(\cdot)$ is an atomic bias assignment for the atoms in B and if $\cdot \Uparrow \cdot \vdash B \Uparrow \cdot$ is provable in LJF then B is provable in LJ.*

The theorem above implies that polarization of atomic formulas does not affect *provability* in *LJF*: in particular, if $\cdot \Uparrow \cdot \vdash B \Uparrow \cdot$ is provable for some atomic bias assignment then that sequent is provable for all such polarizations. On the other hand, different choices of atomic

bias assignments can make a big difference in the shape and size of *LJF* proofs. We illustrate this difference in the next section. Before doing that, we define the *order of a formula* and state two technical results about the order of formulas within *LJF* proofs.

▶ **Definition 3** (Order of a formula). *The* order *of the formula B, written $ord(B)$, is defined as follows: $ord(A) = 0$ if A is atomic and $ord(B_1 \supset B_2) = \max(ord(B_1) + 1, ord(B_2))$.*

Note that if we claim that all formulas in a multiset must have an order that is less than 0, then that multiset must necessarily be empty. The following proposition is proved by a straightforward induction on the structure of formulas.

▶ **Proposition 4.** Let B be a formula such that $ord(B) \leq n$. There is an $\Uparrow$-phase that has $\cdot \Uparrow \cdot \vdash B \Uparrow \cdot$ as its conclusion and has premises that are border sequents. Those border sequents are of the form $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow A$, where A is atomic (i.e., $ord(A) = 0$) and the formulas in Γ have order less than or equal to $n - 1$.

The following proposition is proved by a simple induction on the structure of proofs.

▶ **Proposition 5.** Let Ξ be a (cut-free) *LJF* proof of the border sequent $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow A$, where A is atomic and the formulas in Γ have order less than or equal to n. Then every border sequent in Ξ is of the form $\Gamma, \Delta \Uparrow \cdot \vdash \cdot \Uparrow A'$ where A' is an atomic formula and all formulas in Δ are of order less than or equal to $n - 2$.

2.2 Synthetic inference rules

The following definitions are based on similar definitions in [27].

▶ **Definition 6** (Synthetic inference rule). *A* left synthetic inference rule *is a rule of the form*

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ B. \qquad \textit{This rule is justified by a derivation of the form} \qquad \frac{\begin{array}{c}\Gamma_1 \Uparrow \cdot \vdash \cdot \Uparrow \Delta_1 \quad \ldots \quad \Gamma_n \Uparrow \cdot \vdash \cdot \Uparrow \Delta_n. \\ \Pi \\ \Gamma \Downarrow B \vdash \Delta\end{array}}{\Gamma \Uparrow \cdot \vdash \cdot \Uparrow \Delta}\ D_l$$

Here, $B \in \Gamma$, $n \geq 0$, and within Π, a $\Downarrow$-sequent never occurs above an $\Uparrow$-sequent. The structure of LJF proofs also forces $B \in \Gamma_i$ for all $1 \leq i \leq n$. Given our use of only implications, it is the case that there is a unique left synthetic inference rule for a given formula. The formula B can be used to name this left synthetic inference rule, and we say that this is the left synthetic inference rule for B. *We can similarly define the notion of* right synthetic inference rule*: however, in our context where the only positive formulas that can be used in a D_r rule are atomic, the only inference rule that can be applied to the premise of the D_r rule is I_r. As a result, right synthetic inference rules in our setting have zero premises and appear only at the leaves of proofs. We often write just* synthetic inference rule *to mean the left variant.*

▶ **Definition 7** (Bipole). *A* bipole *is a (left) synthetic inference rule in which all formulas stored using the store rules within the derivation justifying this synthetic inference rule are atomic formulas.*

Bipoles are, therefore, synthetic inference rules in which the only difference between the concluding sequent and any one of its premises is the presence or absence of atomic formulas. Note that, since we are only admitting implications, the synthetic rule for B is a bipole if and only if $ord(B) \leq 2$.

The primary use of the *LJF* proof system in this paper is to build large-scale, synthetic rules that we can add to the unpolarized proof system *LJ*: focusing gives us a framework to create such rules from Gentzen's micro rules. We define such an extension of *LJ* below and then illustrate it with two examples.

▶ **Definition 8** (Rules from polarized theory). *Let $\mathcal{T}$ be a finite set of formulas of order two or less, and let δ be an atomic bias assignment. We define $LJ\lfloor\delta, \mathcal{T}\rfloor$ to be the two-sided proof system built as follows. The only sequents in the $LJ\lfloor\delta, \mathcal{T}\rfloor$ 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 $LJ\lfloor\delta, \mathcal{T}\rfloor$ correspond to synthetic inference rules in the following way. For every left synthetic inference rule*

$$\frac{\mathcal{T}, \Gamma_1 \vdash A_1 \quad \dots \quad \mathcal{T}, \Gamma_n \vdash A_n}{\mathcal{T}, \Gamma \vdash A}\, B, \quad \text{where } B \in \mathcal{T} \text{ is a negative formula, then the rule} \quad \frac{\Gamma_1 \vdash A_1 \quad \dots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A}\, B$$

is added to $LJ\lfloor\delta, \mathcal{T}\rfloor$. The right synthetic inference rules are of the form

$$\frac{}{\Gamma \vdash A} \text{ provided } A \in \mathcal{T} \text{ and } \delta(A) = +,$$

and these are added as well. The only other inference rule added to $LJ\lfloor\delta, \mathcal{T}\rfloor$ is the rule

$$\frac{}{\Gamma, A \vdash A}\ \mathit{init}$$

where A is atomic. If $\delta(A) = -$, this rule is justified by the D_l rule, while if $\delta(A) = +$, this rule is justified by the D_r rule.

▶ **Example 9.** Let $n \geq 0$ and let $a_0, a_1, \dots, a_n$ be a sequence of distinct atomic (propositional) formulas. Let $\mathcal{T}$ be the multiset of formulas $\{d_0, \dots, d_n\}$, where d_0 is a_0, d_1 is $a_0 \supset a_1$, and so on until d_n is $a_0 \supset \cdots \supset a_{n-1} \supset a_n$. Let $\delta^-(\cdot)$ be the atomic bias assignment that gives all atomic formulas the negative polarity and let $\delta^+(\cdot)$ be the atomic bias assignment that gives all atomic formulas the positive polarity. The inference rules in $LJ\lfloor\delta^-, \mathcal{T}\rfloor$ contains the *init* rule along with the following $n + 1$ rules

$$\frac{}{\Gamma \vdash a_0}\, d_0 \qquad \frac{\Gamma \vdash a_0}{\Gamma \vdash a_1}\, d_1 \qquad \frac{\Gamma \vdash a_0 \quad \Gamma \vdash a_1}{\Gamma \vdash a_2}\, d_2 \qquad \cdots \qquad \frac{\Gamma \vdash a_0 \quad \cdots \quad \Gamma \vdash a_{n-1}}{\Gamma \vdash a_n}\, d_n.$$

Given these inference rules, there is a *unique* proof of $\vdash a_n$, and that proof has 2^n occurrences of these inference rules. The negative bias assignment to atomic formulas yields the *backchaining* interpretation of these formulas. In contrast, the inference rules in $LJ\lfloor\delta^+, \mathcal{T}\rfloor$ contains the *init* rule along with the following n rules

$$\frac{\Gamma, a_0, a_1 \vdash A}{\Gamma, a_0 \vdash A}\, d_1 \qquad \frac{\Gamma, a_0, a_1, a_2 \vdash A}{\Gamma, a_0, a_1 \vdash A}\, d_2 \qquad \cdots \qquad \frac{\Gamma, a_0, \dots, a_{n-1}, a_n \vdash A}{\Gamma, a_0, \dots, a_{n-1} \vdash A}\, d_n.$$

Given these inference rules, it is easy to note that there are an infinite number of proofs of $a_0 \vdash a_n$, and that the shortest of these proofs contains n occurrences of synthetic inference rules plus one occurrence of the initial rule. The positive bias assignment to atomic formulas yields the *forward-chaining* interpretation of these clauses.

There are several proofs in the literature that show how cut can be eliminated within focusing proofs. Some proofs, such as the one given by Liang & Miller in [24, 26], introduce different variants of the cut rule and follow a rather tedious argument detailing how these cut

rules move through individual inference rules within ⇑ and ⇓ phases. Bruscoli & Guglielmi [4] provided a different style of proof of cut elimination in a focused proof system for linear logic in which they showed how cuts could move through entire phases at a time. Other phase-based cut-elimination proofs appear in [6, 25, 35, 41, 42]. Part II of Graham-Lengrand's HdR dissertation [20] and Simmons's article [37] provide overviews of such cut-elimination proofs. While none of the proofs mentioned above deal directly with *synthetic* rules in the sense that we have defined them here, the paper [27] does have a cut-elimination theorem that directly deals with eliminating cuts in proofs with synthetic rules. The following theorem follows directly from [27, Theorem 16]. It can also be proved directly by simple induction. Let the *atomic cut rule* be the rule

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\ \mathit{Acut},$$

where A is atomic. When we add this rule to $LJ\lfloor\delta, \mathcal{T}\rfloor$, then B is also atomic.

▶ **Theorem 10** (Cut admissibility for $LJ\lfloor\delta, \mathcal{T}\rfloor$). *Let $\mathcal{T}$ be a set of formulas of order two or less and let $\delta(\cdot)$ be an atomic bias assignment. The atomic cut rule is admissible for the proof system $LJ\lfloor\delta, \mathcal{T}\rfloor$.*

3 Synthetic inference rules as term constructors

Many approaches to relating proofs and terms start with the assumption that terms have been defined, say, as λ-term, and that the notion of proof is derived from terms, e.g., proofs are often identified as dependently typed λ-terms [21]. As we motivated in the introduction, we plan to reverse this influence by starting with proof structures as given and then deriving term structures from them. In particular, we shall map synthetic inference rules directly to the constructors used to build terms.

The first step in translating proofs to terms is to annotate sequents properly. We shall annotate assumptions (i.e., formulas on the left-hand side of sequents) with variables (i.e., token of type *var*) and annotate the conclusion (i.e., the formula on the right-hand side of sequents) with a term of type *tm*. The term constructor that corresponds to the *init* rule (Definition 8) is written $\lceil\cdot\rceil$ and has syntactic type $\mathit{var} \to \mathit{tm}$. The corresponding annotated inference rule is simply

$$\frac{}{\Gamma \vdash \lceil x \rceil : A}\ \mathit{init}, \text{ provided that } x : A \in \Gamma.$$

The following example illustrates how the proofs in Example 9 can be seen as terms.

▶ **Example 11.** Consider changing the naming of the formulas used in Example 9 into typing assumptions, as follows:

$$d_0 : a_0, \quad d_1 : a_0 \supset a_1, \quad \ldots, \quad d_n : a_0 \supset \cdots \supset a_{n-1} \supset a_n.$$

Here, $d_0, \ldots, d_n$ are all of syntactic type *var*. Under the δ^- atomic bias assignment, the synthetic rules can be annotations as

$$\frac{\Gamma \vdash t_0 : a_0 \quad \cdots \quad \Gamma \vdash t_{i-1} : a_{i-1}}{\Gamma \vdash (E_i\ t_0\ \cdots t_{i-1}) : a_i}\ d_i,$$

where $0 \le i \le n$. The syntactic type of constructor E_i is $\mathit{tm} \to \cdots \to \mathit{tm} \to \mathit{tm}$ (where *tm* occurs $i+1$ times). On the other hand, if all atomic formulas are polarized positively, then the annotated synthetic rules are

$$\frac{\Gamma,\ x_0 : a_0,\ \ldots,\ x_{i-1} : a_{i-1},\ y : a_i \vdash t : A}{\Gamma,\ x_0 : a_0,\ \ldots,\ x_{i-1} : a_{i-1} \vdash (F_i\ x_0\ \ldots\ x_{i-1}\ (\lambda y.t)) : A} \quad \text{(provided } y \text{ is new)},$$

where $i \geq 1$. The syntactic type of F_i $(i \geq 1)$ is $var \to \cdots \to var \to (var \to tm) \to tm$ (where var occurs $i+1$ times). Below, we display the unique proof of a_4 using the E_n constructors and the shortest proof of a_4 using the F_n constructors: the structure on the right allows for explicit sharing of subterms while the structure on the left must repeat these subterms.

$$(E_4\ (E_3\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))\ (E_2\ (E_1\ E_0)\ (E_1\ E_0)))\\ \quad (E_3\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))))$$

$$\begin{array}{ll} (F_1\ d_0 & (\lambda x_1.\\ (F_2\ d_0\ x_1 & (\lambda x_2.\\ (F_3\ d_0\ x_1\ x_2 & (\lambda x_3.\\ (F_4\ d_0\ x_1\ x_2\ x_3 & (\lambda x_4.\ \lceil x_4 \rceil)))))))) \end{array}$$

More generally, we can describe the structure of synthetic inference rules as follows. In order to compute the left synthetic inference rule that corresponds to using D_l on the indexed formula $f : B$, where $ord(B) \leq 2$, we must know how the atomic formulas in B are polarized. In what follows, we assume that there are exactly two atomic bias assignments, namely, δ^- and δ^+. We consider these two cases separately. In the following discussion, we use Δ as a schematic variable to range over multisets of atomic formulas and use $\bar{w}$ to denote a list of variables, say, $w_1, \ldots, w_n$ for some $n \geq 0$. The notation $\bar{w} : \Delta$ denote a type assignment, say, $w_1 : A_1, \ldots, w_n : A_n$, where $\bar{w}$ is $w_1, \ldots, w_n$ and Δ is $A_1, \ldots, A_n$. Finally, whenever we construct typing assignments for the left-hand side of sequents, we will always assume that the names assigned types are all distinct.

Consider first the case that we are using δ^-. Any formula B such that $ord(B) \leq 2$ can be written as

$$(\Delta_1 \supset A_1) \supset \cdots \supset (\Delta_m \supset A_m) \supset A_0 \qquad (m \geq 0)$$

where $\Delta_1, \ldots, \Delta_m$ are multisets of atomic formulas. (If Δ is empty then $\Delta \supset B$ is simply B.) If the displayed formula above has order 0 then $m = 0$. If that formula has order 1 then $m \geq 1$ and $\Delta_1, \ldots, \Delta_m$ are empty. If that formula has order 2 then $m \geq 1$ and at least one of $\Delta_1, \ldots, \Delta_m$ is nonempty. In any case, all of the formulas $\Delta_1 \supset A_1, \ldots, \Delta_m \supset A_m$ have negative polarity. In this case, the left synthetic inference rule corresponding to B can be written as

$$\frac{\Gamma, \bar{w}_1 : \Delta_1 \vdash t_1 : A_1 \quad \ldots \quad \Gamma, \bar{w}_m : \Delta_m \vdash t_m : A_m}{\Gamma \vdash (f\ (\lambda \bar{w}_1.t_1) \ldots (\lambda \bar{w}_m.t_m)) : A_0}$$

By $\bar{w} : \{D_1, \ldots, D_n\}$ we mean $w_1 : D_1, \ldots, w_n : D_n$, assuming that all the tokens $w_1, \ldots, w_n$ are all distinct and do not occur in Γ. If the list of variables $\bar{w}$ is empty, then we abbreviate $\lambda \bar{w}.t$ as simply t.

Now consider the second case where we are using δ^+. In this case, we note that any formula B such that $ord(B) \leq 2$ can be written as

$$\Delta_0 \supset (\Delta_1 \supset A_1) \supset \cdots \supset (\Delta_m \supset A_m) \supset A_0 \qquad (m \geq 0)$$

(We assume that all the atomic argument types are listed before the non-atomic argument types.) Here, $\Delta_0, \ldots, \Delta_m$ are all multisets of atomic formulas, and we assume that $\Delta_1, \ldots, \Delta_m$ are nonempty. If the displayed formula above has order 0 then $m = 0$ and Δ_0 is empty. If that formula has order 1 then $m = 0$ and Δ_0 is nonempty. If that formula has order 2 then $m \geq 1$. In this way of presenting the type of f, the (atomic) argument types in Δ_0 all have positive polarity, while the remaining argument types all have negative polarity. Thus, the left synthetic inference rule corresponding to B can be written as

$$\frac{\Gamma, \bar{w}_1 : \Delta_1 \vdash t_1 : A_1 \quad \ldots \quad \Gamma, \bar{w}_m : \Delta_m \vdash t_m : A_m \quad \Gamma, y : A_0 \vdash t : A}{\Gamma \vdash \textbf{name}\ y = (f\ \bar{u}\ (\lambda \bar{w}_1.t_1) \ldots (\lambda \bar{w}_m.t_m))\ \textbf{in}\ t : A} \text{ provided } \bar{u} : \Delta_0 \subseteq \Gamma$$

The proviso to this rule means that every type assumption $u_1 : D_1, \ldots, u_n : D_n$ is a member of Γ (here, $\bar{u}$ is $u_1, \ldots, u_n$ and Δ_0 is $\{D_1, \ldots, D_n\}$). We do not assume that the variables in $u_1, \ldots, u_n$ are distinct. The first m premises above construct abstracted terms, and the last premise continues to construct a term for type A but this time with an additional variable y that names the structure $(f\ \bar{u}\ (\lambda\bar{w}_1.t_1) \ldots (\lambda\bar{w}_m.t_m))$. Note also that the first arguments of f (written here as $\bar{u}$) must already be present on the left-hand side of the conclusion.

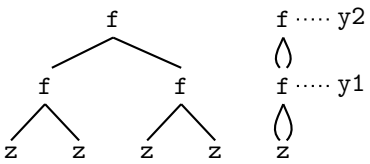
▶ **Example 12.** Using this notation, the last term in Example 11 can be written as follows.

name $x_1 = (F_1\ d_0)$ **in**
name $x_2 = (F_2\ d_0\ x_1)$ **in**
name $x_3 = (F_3\ d_0\ x_1\ x_2)$ **in**
name $x_4 = (F_4\ d_0\ x_1\ x_2\ x_3)$ **in** $\lceil x_4 \rceil$

To provide a simple, graphical illustration of how atomic bias assignments can affect term structure, consider the typing assignment $\{z : i,\ f : i \supset i \supset i\}$. The following two terms denote proofs built with synthetic inference rules in which i has a negative and positive polarity, respectively.

```
(f (f z z) (f z z))
name y1 = (f z z) in name y2 = (f y1 y1) in y2.
```

These terms could also be displayed as the following labeled tree and DAG. Here, we display the tree associated with a term with its root at the top and its leaves below: this is, of course, the opposite convention to how proof trees are commonly displayed.



In an appendix of the extended version of this paper [32], we show how these constructors – corresponding to synthetic inference rules – can be implemented using the $\lambda\kappa$-calculus, a term representation for *LJF* introduced by Brock-Nannestad et al. in [3].

In the following subsections, we note two applications of the term representations that we have described above.

3.1 Intermediate representation of programs

The `name` expressions used above resemble the more common `let` expressions, but they are different in at least two important ways. First, the let-expression "let $x = r$ in t" is often considered to be a non-normal term since it sometimes abbreviates the β-redex $((\lambda x.t)r)$. In contrast, the term "name $x = r$ in t" is normal if t is normal. (Note that the synthetic inference rules are built without the use of the cut inference rule.) Second, the term r in a let-expression can be an arbitrary term, whereas in the name-expression, the term r must have a particular structure: when we are considering only first-order signatures, r must be the application of a constructor to one or more variables (and not to general terms).

Given an expression E using negative bias syntax, there might be a subexpression, say E', that has many occurrences in E. If we let $F(x)$ denote the result of replacing every occurrence of E' in E with the variable x, then the expression $let\ x = E'\ in\ F(x)$ might be

a more appropriate presentation of E in which the subformula E' is named and explicitly shared. Such an operation in often called *common subexpression elimination*. The positive bias syntax that we have described here is orthogonal to this processing in the sense that the positive bias assignment syntax forces *all* function applications (not just those that are repeated) to be named while there is no guarantee that naming is not redundant (i.e., a given subterm might be given two names). Of course, if one takes care in building terms using positive bias assignment, it is possible to build terms where common subexpressions are explicitly shared.

A useful intermediate representation of programs in compilers of functional programming languages is the *administrative normal form (ANF)* [13] in which all arguments to functions are values, that is, either constants, variables, or λ-abstractions. Clearly, when we are using positive bias assignment syntax, the expressions that result are in ANF.

Various other term representations have been developed for focused proofs that contain more logical connectives and inference rules than we have considered here. See, for example, [5, 23, 37]. However, atomic formulas are given the negative polarity in these references.

3.2 Encoding functional expressions as relational queries

When a Prolog programmer needs to compute the value of a mathematical expression, such as $\sqrt{b^2 - 4ac}$, it is necessary to explicitly convert the calls to various functions (here, subtraction, addition, multiplication, and square root) into associated relations. For example, addition on real numbers is usually represented by the three-place predicate *plus* such that the atomic formula $(plus\ x\ y\ z)$ holds if and only if $x + y = z$. Now assume that relations are available to encode each primitive function. One way to organize the relations needed to compute the expression above involves converting that expression into positive bias syntax. For example, the function-based expression above can be written as

$$\textbf{name}\ n_1 = b \times b\ \textbf{in name}\ n_2 = 4 \times a\ \textbf{in name}\ n_3 = n_2 \times c\ \textbf{in}$$
$$\textbf{name}\ n_4 = n_1 - n_3\ \textbf{in name}\ n_5 = \sqrt{n_4}\ \textbf{in}\ \lceil n_5 \rceil.$$

As described in [16], it is straightforward to convert such an expression into a series of calls to predicates. In particular, we can rewrite this expression by replacing $[\textbf{name}\ n = f\ x_1\ \cdots\ x_i\ \textbf{in}\ \bullet]$ with $[\exists n.(R_f\ x_1\ \cdots\ x_i\ n) \wedge \bullet]$, where R_f is a relation that computes the function f. Assuming that *times*, *minus*, and *sqrt* are all relations that compute multiplication, subtraction, and the (positive) square root, then the relational presentation can be given as

$$\exists n_1.\, times\ b\ b\ n_1 \wedge \exists n_2.\, times\ 4\ a\ n_2 \wedge \exists n_3.\, times\ n_2\ c\ n_3 \wedge \exists n_4.\, minus\ n_1\ n_3\ n_4 \wedge \exists n_5.\, sqrt\ n_4\ n_5,$$

which is an expression that is easily written as a Prolog goal formula.

4 The untyped lambda-calculus

Let D be an atomic formula and let Γ_0 be the theory $\{(D \supset D) \supset D,\ D \supset (D \supset D)\}$. This theory is *inconsistent* in that every formula built from implications and D is provable from Γ_0. We choose to consider Γ_0 because cut-free proofs in *LJF* of $\Gamma_0 \vdash D$ correspond to closed untyped λ-terms. The following derivations result from applying D_l to these two formulas (assuming that $\Gamma_0 \subseteq \Gamma$).

$$\dfrac{\dfrac{\Xi_1 \quad \Xi_2 \quad \Xi_3}{\Gamma \Downarrow D \supset (D \supset D) \vdash D}\ \supset L \times 2}{\Gamma \vdash D}\ D_l \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Xi_4}{\Gamma, D \vdash D}}{\Gamma \Uparrow D \vdash D \Uparrow}\ S_l, S_r}{\Gamma \vdash D \supset D \Uparrow}\ R_r, \supset R \quad \dfrac{\Xi_5}{\Gamma \Downarrow D \vdash D}}{\Gamma \Downarrow (D \supset D) \supset D \vdash D}\ \supset L}{\Gamma \vdash D}\ D_l$$

These derivations can only be extended to have border sequents as premises if we reveal the polarity assigned to D. If D is polarized negatively, then Ξ_1 and Ξ_2 are both R_r while Ξ_3 and Ξ_5 are I_l. In this case, the resulting synthetic inference rules are

$$\frac{\Gamma \vdash D \qquad \Gamma \vdash D}{\Gamma \vdash D} \quad \text{and} \quad \frac{\Gamma, D \vdash D}{\Gamma \vdash D} \ .$$

On the other hand, if D is polarized positively, then Ξ_1 and Ξ_2 are both I_r while Ξ_3 and Ξ_5 is R_l. In this case, the resulting synthetic inference rules are

$$\frac{\Gamma, D, D, D \vdash D}{\Gamma, D, D \vdash D} \quad \text{and} \quad \frac{\Gamma, D \vdash D \qquad \Gamma, D \vdash D}{\Gamma \vdash D} \ .$$

Without annotations, these inference rules are not illuminating. We provide such annotations next.

4.1 Using negative bias syntax for the untyped lambda-calculus

If we polarize D negatively, the (annotated) synthetic inference rules based on Γ_0 yield the standard tree-like representation for the untyped λ-calculus. In particular, consider an annotated LJF proof of a sequent of the form $\Gamma_0, x_1 : D, \ldots, x_n : D \vdash t : D$. The synthetic inference rules in such a proof are either the result of deciding on $x_i : D$ (for $1 \leq i \leq n$) followed by I_l or on one of the two formulas in Γ_0. The three inference rules annotate these three choices for using the D_l rule.

$$\frac{\Gamma \vdash t : D \qquad \Gamma \vdash s : D}{\Gamma \vdash \mathit{napp}\ t\ s : D} \qquad \frac{\Gamma, x : D \vdash t : D}{\Gamma \vdash \mathit{nabs}\ (\lambda x.t) : D} \qquad \frac{}{\Gamma \vdash \mathit{nvar}\ x_i : D} \text{ provided } x_i : D \in \Gamma$$

The constructors represented by these synthetic rules are listed below, along with their syntactic types (using typing declarations from λProlog [29]).

```
type napp   tm -> tm -> tm.
type nabs   (var -> tm) -> tm.
type nvar   var -> tm.
```

As we have assumed before, `var` and `tm` are primitive types: annotations on the left side of sequents have type `var` while annotations on the right side have type `tm`. The prefix `n` on these names is meant to remind us that we assigned D the negative polarity. As an example of using these constructors, the untyped λ-term $((\lambda x.xx)(\lambda x.xx))$ can be written as the following term of type tm:[1]

```
(napp (nabs x\ napp (nvar x) (nvar x))
      (nabs x\ napp (nvar x) (nvar x))).
```

Many computational logic systems, such as λProlog, Abella, LF, and Isabelle, encode the untyped λ-calculus in this fashion.

4.2 Using positive bias syntax for the untyped lambda-calculus

If we polarize D positively, we get different synthetic rules based on using Γ_0 and, hence, we get a different format for encoding the untyped λ-calculus. Again, there are exactly three synthetic inference rules for LJF proofs of sequents of the form $\Gamma_0, x_1 : D, \ldots, x_n : D \Uparrow \cdot \vdash \cdot \Uparrow t : D$, but this time there are two left synthetic inference rules and one right synthetic inference rules.

[1] The λ-abstraction of λProlog is written using an infix backslash.

$$\frac{\Gamma, y : D \vdash t : D}{\Gamma \vdash \mathit{papp}\ x_i\ x_j\ (\lambda y.t) : D} \text{ provided } \Gamma \text{ contains } x_i : D \text{ and } x_j : D$$

$$\frac{\Gamma, x : D \vdash t : D \qquad \Gamma, y : D \vdash s : D}{\Gamma \vdash \mathit{pabs}\ (\lambda x.t)\ (\lambda y.s) : D} \qquad \frac{}{\Gamma \vdash \mathit{pvar}\ x_i : D} \text{ provided } x_i : D \in \Gamma$$

The constructors represented by these synthetic rules can be given the following syntactic typing.

```
type papp  var -> var -> (var -> tm) -> tm.
type pabs  (var -> tm) -> (var -> tm) -> tm.
type pvar  var -> tm.
```

The prefix `p` on these names is meant to remind us that, in this case, we assigned D the positive polarity. Using these constructors, the untyped λ-term $((\lambda x.xx)(\lambda x.xx))$ can be written as the term

```
(pabs (x\ papp x x (y\ pvar y)) (u\ papp u u (z\ pvar z)))
```

Note that this untyped λ-term contains two occurrences of `papp` while in the previous representation, this term contains three occurrences of `napp`.

In order to make terms in this syntax easier to read, we will often use the `name`-expressions presented before. In particular, the expression `(papp u v (w\ Body))`, which denotes the application of the variable `u` to the variable `v`, and then *naming* that application as `w` in the scope of `Body`. Thus, this expression can also be written using the expression `name w = (app u v) in Body`. Similarly, the expression `(pabs R (w\ Body))` can be written as `name w = (abs R) in Body`. If we used this syntax, then the expression above denoting $((\lambda x.xx)(\lambda x.xx))$ is written as

```
  name u = (abs x\ name y = (app x x) in y) in
  name z = (app u u) in z
```

That is, `u` is used to name the encoding for the term $(\lambda x.xx)$, and then that name is used twice in building the final application that is named `z`. Extending this example slightly, we see that the expression $(((\lambda x.xx)(\lambda x.xx))(\lambda x.xx))$ can be written as

```
(pabs (x\ papp x x (y\ pvar y)) (u\ papp u u (z\ papp z u v\ pvar
    v)))
```

or, using the `name` syntax, as

```
  name u = (abs x\ name y = (app x x) in y) in
  name z = (app u u) in
  name v = (app z u) in v.
```

As this alternative syntax suggests, the syntax that results from making the primitive type D positive makes sharing explicit by its requirement that all applications are built from *named* structures and that that application is named itself. It is also clear that the following two expressions denote the same untyped λ-term.

```
  name u  = (abs x\ name y = (app x x) in y) in
  name z1 = (app u u) in
  name z2 = (app u u) in
  name v  = (app z1 u) in v
```

```
name u1 = (abs x\ name y = (app x x) in y) in
name z  = (app u1 u1) in
name u2 = (abs x\ name y = (app x x) in y) in
name v  = (app z u2) in v
```

The first of these terms illustrates that a named structure might not be used in its scope: we call this *vacuous naming*. The second of these terms illustrates that the same structure can be named twice: we call this *redundant naming*. The proof theory behind *LJF* allows for both vacuous and redundant naming: we currently see no proof-theoretic device that can cleanly eliminate these kinds of naming expressions.

This style of syntactic representation seems relatively low-level since it uses names to designate all constructors in a term. Such a representation of terms resembles, in fact, the use of pointers to encode terms in memory: a pointer (name) indicates a unit of memory that contains the name of a constructor followed by a vector of pointers to that constructor's arguments.

4.3 Tracing untyped lambda-terms

Given that we have two different formats for untyped λ-terms, it is a natural question whether or not two such expressions denote the same untyped λ-term. For example, it seems sensible to consider the last three expressions in Section 4.2 (based on positive bias assignment) as equivalent in some sense to each other and to the following expression (based on negative bias assignment).

```
(napp (napp (nabs x\ napp (nvar x) (nvar x))
            (nabs x\ napp (nvar x) (nvar x)))
      (nabs x\ napp (nvar x) (nvar x)))
```

Broadly speaking, there are two approaches to answering this question. The "white box" approach examines the actual syntax of proof expressions to see if they should be considered equal. For example, in the setting of natural deduction, two proofs are often considered equal if they reduce to the same normal form. Given that we are considering proofs built with difference sets of (synthetic) inference rules, a different approach needs to be taken, such as the approach described in [35], where proofs based on positive bias assignment to atomic formulas are systematically converted to proofs based on negative bias assignment.

Instead, we propose to use a "black box" approach in which we probe a term to describe *traces* within expressions. For example, we can ask whether or not the term denotes a top-level application or not. This check is easy to make for negative bias syntax by simply examining the top-level symbol of the expression. It is also easy to check for the expressions using the positive bias assignment: simply examine the top-level naming structure, say,

$$[\texttt{name}\ x_1 = E_1\ \texttt{in}\ \texttt{name}\ x_2 = E_2\ \texttt{in}\ \cdots\ \texttt{name}\ x_n = E_n\ \texttt{in}\ x_j]$$

and check if E_j is an application or not. If two expressions denote an application, we can continue to develop a trace by examining either the first or second argument of that application. Similarly, we can examine two expressions to see if they denote a λ-abstraction. If they are both λ-abstractions, then we can probe the body of those abstractions, taking appropriate care when descending under a binding.

A formal specification of such trace predicates is easy in a language such as λProlog. In particular, the following declarations define the datatype of traces through untyped λ-terms.

```
type ntrace, ptrace   tm -> trace -> o.
```

```
ntrace (napp M _) (left  P) :- ntrace M P.
ntrace (napp _ N) (right P) :- ntrace N P.
ntrace (nabs R)   (bnd   P) :- pi x\pi p\ ntrace (nvar x) p =>
                                          ntrace (R x) (P p).

ptrace (papp U V K) P :-
   pi x\ (pi P\ ptrace (pvar x) (left  P) :- ptrace (pvar U) P) =>
         (pi P\ ptrace (pvar x) (right P) :- ptrace (pvar V) P) =>
   ptrace (K x) P.
ptrace (pabs R K) P :-
   pi x\  (pi Q\ ptrace (pvar x) (bnd Q) :-
                 pi p\ pi u\ ptrace (pvar u) p => ptrace (R u) (Q
                     p))
          => ptrace (K x) P.
```

Figure 2 Traces through negative and positive bias syntax.

```
kind trace          type.
type left, right    trace -> trace.
type bnd            (trace -> trace) -> trace.
```

The specification of traces within both variants of expressions for (closed) untyped λ-terms is given in Figure 2. Note that the orders of the clauses for `ntrace` are 0, 1, 2 while for `ptrace` the orders are 0, 3, and 4.[2] If we wish to treat open expressions, we can add a constant, say `w`, to denote a free variable, along with the declarations

```
type w       var.
type wtrace  trace.
```

```
ntrace  (nvar w) wtrace.
ptrace  (pvar w) wtrace.
```

We say that two expressions denoting untyped λ-terms (using either positive or negative bias assignment) are *trace equivalent* if they both have the same traces.[3] It is easy to prove that two expressions using the negative bias syntax are trace equivalent if and only if they are α-equivalent.[4] This statement is not true for positive bias syntax: in particular, the examples at the end of Section 4.2 that illustrate vacuous and duplicate naming all have the same traces but are not α-convertible expressions.

We note that in λProlog, it is possible to synthesize an expression from a list of traces using, for example, a query such as

```
?- forall (ntrace T) [(bnd (u\ left (bnd (v\ left u)))),
                      (bnd (u\ left (bnd (v\ right v)))),
                      (bnd (u\ right u))].
T = nabs (u\ napp (nabs (v\ napp (nvar u) (nvar v))) (nvar u))
```

[2] Standard techniques can be used to rewrite the last two of these clauses to clauses of order 2 at the expense of adding new predicate constants. See Appendix A of [32] for such a specification.
[3] In concurrency theory, this notion is more often called *maximal trace equivalence*.
[4] See `http://abella-prover.org/examples/lambda-calculus/term-structure/path.html` for a short, formal proof of this claim in Abella.

```
Kind op       type.
Type app      val  -> val -> op.
Type abs      (val -> tm) -> op.

Kind pair     type.
Type pr       val -> op -> pair.

Kind node     type.
Type nd       list pair -> tm -> node.
```

Figure 3 An Abella specification of sharing simulation.

Here, `forall` is a higher-order predicate that applies its predicate argument (here, `(ntrace T)`) to all members in its second argument. A black box method of converting an expression using the positive bias assignment into an expression using negative bias assignment proceeds as follows: first, list all possible traces in the positive bias assignment expression, and second, synthesize the negative bias assignment expression using the technique illustrated above (see also [29, Section 7.4.2]).

4.4 Sharing bisimulation

Determining that two untyped λ-term expressions are trace equivalent by enumerating every trace in them has an exponential cost since all sharing structures are removed when listing traces. Condoluci et al. [9] developed a graphic representation of sharing in the untyped λ-calculus using λ*graphs*. When an appropriate bisimulation is defined on nodes in such λgraphs, it is possible to check the bisimilarity in such graphs in linear time. As is known from concurrency theory, bisimilarity implies (maximal) trace equivalence. In our setting, if two terms – represented as two nodes in a λgraph – are bisimilar, then those two terms are also trace equivalent.

To illustrate how one can manipulate positive bias syntax effectively, we used Abella [2] to specify a simulation relation (closely related to the bisimulation relation defined in [9]) that compares two expressions in such a way that unfolding of the sharing does not happen.

The top-level of an untyped λ-term expression based on the positive bias assignment for D can be described as a pair containing an association list of naming variables and operations (such as `app` and `abs` used above) and the name of a particular naming variable. For example, the last term displayed in Section 4.2 can be seen as a list of four pairs, namely,

$$[\ \langle \texttt{u1}, \texttt{(abs x name y = (app x x) in y)}\rangle,\ \langle \texttt{z}, \texttt{(app u1 u1)}\rangle,$$
$$\langle \texttt{u2}, \texttt{(abs x name y = (app x x) in y)}\rangle,\ \langle \texttt{v}, \texttt{(app z u2)}\rangle\ ].$$

along with the designation of one particular variable, here `v`. This presentation suggests encoding such terms using the Abella declarations (which are similar to λProlog declarations) found in Figure 3. For example, the example term displayed above can be written as the Abella term of type `name`

```
(nd ((pr n4 (app n2 n3)) ::
     (pr n3 (abs x\ papp x x y\ pvar y)) ::
     (pr n2 (app n1 n1)) ::
     (pr n1 (abs x\ papp x x y\ pvar y)) :: nil)
    (pvar n4))
```

(Here, the symbols `n1`, ..., `n4` are examples of nominal constants in Abella.) In this setting, expressions using positive bias syntax are encoded as terms of type `node`.

Structures of type `node` can be used to generate a labeled transition system in which some arcs are labeled. These labels, called *actions* here, are of the following three kinds (see the full specification in Figure 4).

1. Primitive actions are described using the predicate `paction`. Such actions name a variable.
2. Bound actions, specified using `baction`, carry an abstraction node to the body of that abstraction.
3. Application actions, specified using `faction`, carry an application node to either its left or right argument: this action names that direction.

The ∇-quantifier [14, 31] (written in Abella as `nabla`) is used to manage nominal constants and binding mobility [28]. The distinction between *free action* and *bound action* here is essentially the same as has been used to specify various simulations in the π-calculus [33]. Our specification of simulation in the presence of both free and bound actions follows the specification technique of Tiu & Miller [38] that also relied on the ∇-quantifier. Given our simulation specification, the bisimulation specification is also easy to write.

5 Cut elimination at the level of synthetic rules

Theorem 10 states that cut elimination holds for proofs built using synthetic inference rules. Our goal in this section is to use cut elimination to determine what substitution into terms should be. In particular, if we have the term t and we have the abstraction of x over term s, how do we compute the result of substituting t for x in s? Clearly, the answer will depend on which polarity assumption we are using for primitive types.

In order to see how cut-elimination can yield substitution, consider the following instance of the cut rule: here, E and E' are atomic formulas, and the LJF proofs Ξ_L and Ξ_R are cut-free.

$$\frac{\begin{array}{cc}\overset{\Xi_L}{\Gamma \vdash t : E'} & \overset{\Xi_R}{\Gamma, x : E' \vdash s : E}\end{array}}{\Gamma \vdash Cut_0(x.s, t) : E}\, Cut_0$$

While the term $Cut_0(x.s, t)$ is not a term, it denotes the result of substituting u for x in t. By performing cut-elimination on this proof, we will arrive at a cut-free proof and the term annotating that proof should denote the result of such a substitution.

We illustrate here how the cut-elimination procedure works on our two encodings of untyped λ-terms.[5] In particular, we will provide specifications (using λProlog code) to define the predicates

```
type nsubst, psubst    tm -> (var -> tm) -> tm -> o.
```

that have the following specification: given `S` and `T` of type `tm` and `R` of type `var -> tm` then `(nsubst T R S)` is provable if and only if `T`, `R`, and `S` use the constructors `napp`, `nabs`, and `nvar` and `S` is the result of substituting `T` into the bound variable of `R`. Similarly, we wish to have the same kind of specification for `(psubst T R S)` but with the arguments `S`, `T`, and `R` built using the constructors `papp`, `pabs`, and `pvar`.

[5] The cut-elimination procedure is given in Appendix B of [32].

```
Define paction : list val -> node -> val -> prop by
  paction Vs (nd C (pvar w)) w ;
  paction Vs (nd C (pvar V)) V := member V Vs.

Define baction : node -> (val -> node) -> prop by
  nabla n, baction (nd ((pr n (abs R)):: C)  (pvar n))
                   (u\ nd C (R u)) ;
  nabla n, baction (nd ((pr M (Op n)):: (C n)) (pvar n)) Nd :=
     nabla n, baction (nd (C n) (pvar n)) Nd.

Kind direction     type.
Type right, left   direction.

Define faction : node -> direction -> node -> prop by
  nabla n, faction (nd ((pr n (app U V)):: C) (pvar n)) right
                   (nd C (pvar V)) ;
  nabla n, faction (nd ((pr n (app U V)):: C) (pvar n)) left
                   (nd C (pvar U)) ;
  nabla n, faction (nd ((pr M (Op n)):: (C n)) (pvar n)) A T :=
     nabla n, faction (nd (C n) (pvar n)) A T.

Define sim  : list val -> node -> node -> prop,
       simm : list val -> node -> node -> prop by
 sim Vs (nd C (papp U V K)) Nd :=
    nabla n, sim Vs (nd ((pr n (app U V)):: C) (K n)) Nd ;
 sim Vs (nd C (pabs R K)) Nd :=
    nabla n, sim Vs (nd ((pr n (abs R)):: C) (K n)) Nd ;
 sim Vs (nd D (pvar T)) (nd C (papp U V K)) :=
    nabla n, sim Vs (nd D (pvar T)) (nd ((pr n (app U V)):: C) (K
       n)) ;
 sim Vs (nd D (pvar T)) (nd C (pabs R K)) :=
    nabla n, sim Vs (nd D (pvar T)) (nd ((pr n (abs R)):: C) (K
       n)) ;
 sim Vs (nd C (pvar U)) (nd D (pvar V)) :=
   simm Vs (nd C (pvar U)) (nd D (pvar V)) ;

 simm Vs P Q :=
  (forall N,   paction Vs P N -> paction Vs Q N)         /\
  (forall A R, faction P A R  -> exists S, faction Q A S /\
                                           sim Vs R S)   /\
  (forall R,   baction P R    -> exists S,
                                   baction Q S  /\
                                   nabla u, sim (u::Vs) (R u) (S
                                       u)).
```

Figure 4 An Abella specification of sharing simulation.

When terms are built using negative bias syntax, the cut always moves to the right branch, which means that substitution can be defined recursively on `R`. Moreover, substitution is applied to all the arguments of constructors recursively. Thus, we have the following λProlog specification of `nsubst`.

```
nsubst T (x\ nvar x) T.
nsubst T (x\ nvar Y) (nvar Y).
nsubst T (x\ napp (R x) (S x)) (napp R' S') :-
   nsubst T R R', nsubst T S S'.
nsubst T (x\ nabs y\ R x y) (nabs y\ R' y) :-
   pi y\ nsubst T (x\ R x y) (R' y).
```

Note that this substitution predicate moves recursively through its second (abstracted) argument. This style of substitution is, of course, the familiar one. Using the following functional equations, we can write this substitution operation as a postfix operator.

- $(\texttt{nvar}\ x)[x/t] = t$;
- $(\texttt{nvar}\ y)[x/t] = (\texttt{nvar}\ y)$, provided x and y are different;
- $(\texttt{napp}\ R\ S)[x/t] = (\texttt{napp}\ R[x/t]\ S[x/t])$, and
- $(\texttt{nabs}\ (\lambda y.R))[x/t] = (\texttt{nabs}\ \lambda y.\ (R[x/t]))$, provided that x and y are different and y is not free in t.

When terms are built using the positive polarity, the cut moves to the left branch, which means that the substitution can be defined recursively on the first argument.

```
psubst (papp U V K) R (papp U V H) :- pi x\ psubst (K x) R (H x).
psubst (pabs S K)   R (pabs S H)   :- pi x\ psubst (K x) R (H x).
psubst (pvar U)     R (R U).
```

Note that the last line of this specification uses a meta-level β-reduction but only to effect a variable renaming substitution. An example query using this last predicate is the following.

```
?- psubst (papp w w y\ papp y y z\ papp z z v\ pvar v)
          (x\ papp x x pvar) R.
R = papp w w (y\ papp y y (z\ papp z z (v\ papp v v u\ pvar u)))
```

Another example of substitution using this style of syntax is given graphically (and using the `name` notation) in Figure 5.

We can instead write this substitution operation as a prefix operator using the following functional equations. Below, the operation $t[x := u]$ denotes the replacement of every free occurrence of x in t by the variable u, provided that no free occurrence of x is in the scope of a binding on u.

- $[(\texttt{papp}\ u\ v\ (\lambda y.K))/x]R = (\texttt{papp}\ u\ v\ (\lambda y.[K/x]R)$, provided x and y are different;
- $[(\texttt{pabs}\ S\ (\lambda y.K))/x]R = (\texttt{pabs}\ S\ (\lambda y.[K/x]R)$, provided x and y are different; and
- $[(\texttt{pvar}\ u)/x]R = R[x := u]$, provided no free occurrence of x is in the scope of a binding on u.

Given our discussion of checking the simulation of two untyped λ-terms in Section 4.4, we know that such terms can be represented as a pair, say, $\langle \Gamma,\ u\rangle$ where Γ is an association list between a variable (of type `var`) and an operation (of type `op`) that indicates the kind of node that variable names (either an application or an abstraction). An equivalent description for substitution can then be given as follow: The result of substituting the term $\langle \Gamma,\ u\rangle$ for x in the term $\langle \Gamma',\ u'\rangle$ is the term $\langle (\Gamma'[x := u]) \sqcup \Gamma,\ u'\rangle$, where $\sqcup$ denotes appending of two lists.

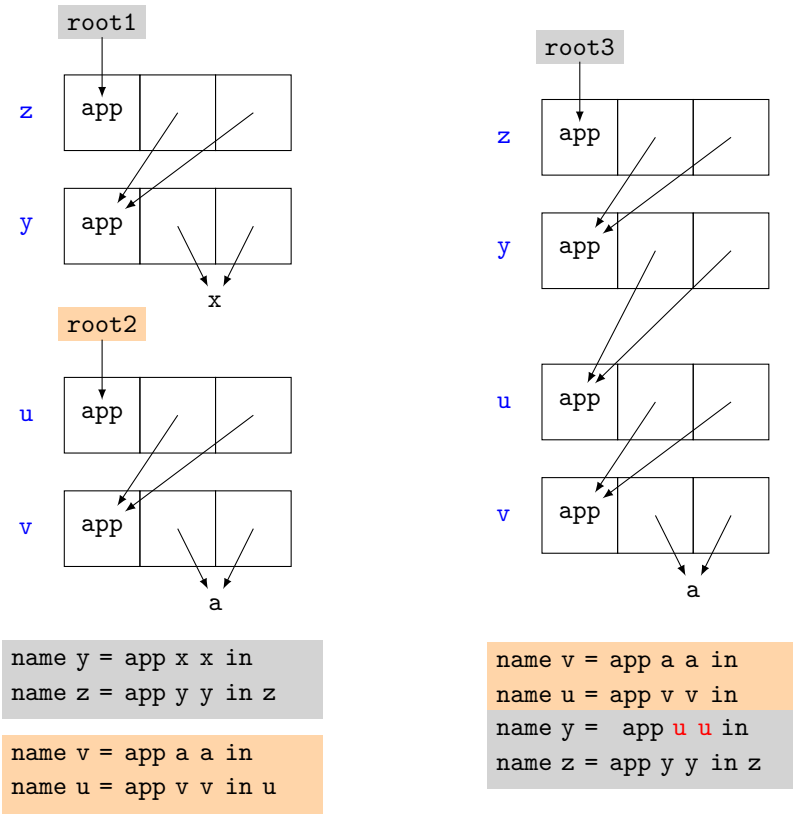


Figure 5 The term at `root3` is the result of substituting the term at `root2` for `x` into the term at `root1`. Operationally speaking, the pointers to `x` are redirected to point to `u` instead.

6 Related and future work

One goal of developing a logical framework, such as *LJF*, is to account for many other calculi within that framework. The literature contains other approaches to term representation that have also been motivated by focused proof systems [11, 12, 23]. Since *LJF* can easily account for several other focusing proof systems for intuitionistic logic [24], this framework should similarly account for such term calculi. Other frameworks have been used to justify term structures: for example, it would be interesting to see if there are any overlaps with the terms-as-graphs work of Grabmayer [19].

While many constructors for building term structures are only second-order, it is natural and occasionally important to be able to treat constructors of order greater than 2. Of course, the proof theory of *LJF* can treat formulas of all orders. However, the notion of synthetic inference rule (which is here limited to second order) would need to be generalized. In a setting with such higher-order constructors, cut elimination should be able to derive and generalize *hereditary substitutions* [40].

As mentioned in Section 4.4, the λgraphs in [9] represent sharing as DAG structures in the untyped λ-calculus. We conjecture that by using a multifocused version of the *LJF* proof system, we should be able to prove that maximal multifocused *LJF* proofs correspond to λgraphs. Maximal multifocused proofs have been shown elsewhere to correspond to graphical proof systems, such as proof nets [8], expansion proofs [7], and natural deduction proofs [35].

The black box methods of probing the structure of terms with sharing (see Sections 4.3 and 4.4) is closely related to well-established results in concurrency theory. However, these methods are not based on proof-theoretic principles, at least not that we have established here. We hope to find a way to describe trace equivalence and bisimilarity via proof-theoretic concepts. These notions seem related to Girard's Ludics project [18]. One promising approach might start with recognizing that simulations can be seen as winning strategies and that winning strategies can be related to focused proofs [10].

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992. doi:10.1093/logcom/2.3.297.

2 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014. doi:10.6092/issn.1972-5787/4650.

3 Taus Brock-Nannestad, Nicolas Guenot, and Daniel Gustafsson. Computation in focused intuitionistic logic. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14–16, 2015*, pages 43–54. ACM, 2015. doi:10.1145/2790449.2790528.

4 Paola Bruscoli and Alessio Guglielmi. On structuring proof search for first order linear logic. *Theoretical Computer Science*, 360(1-3):42–76, 2006. doi:10.1016/j.tcs.2005.11.047.

5 Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003. doi:10.1093/logcom/13.5.639.

6 Kaustuv Chaudhuri. Focusing strategies in the sequent calculus of synthetic connectives. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR: International Conference on Logic, Programming, Artificial Intelligence and Reasoning*, volume 5330 of *LNCS*, pages 467–481. Springer, November 2008. doi:10.1007/978-3-540-89439-1_33.

7 Kaustuv Chaudhuri, Stefan Hetzl, and Dale Miller. A multi-focused proof system isomorphic to expansion proofs. *J. of Logic and Computation*, 26(2):577–603, 2016. doi:10.1093/logcom/exu030.

8 Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In G. Ausiello, J. Karhumäki, G. Mauri, and L. Ong, editors, *Fifth International Conference on Theoretical Computer Science*, volume 273 of *IFIP*, pages 383–396. Springer, September 2008. doi:10.1007/978-0-387-09680-3_26.

9 Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. Sharing equality is linear. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2019. doi:10.1145/3354166.3354174.

10 Olivier Delande, Dale Miller, and Alexis Saurin. Proof and refutation in MALL as a game. *Annals of Pure and Applied Logic*, 161(5):654–672, February 2010. doi:10.1016/j.apal.2009.07.017.

11 Roy Dyckhoff and Stephane Lengrand. Call-by-value λ-calculus and LJQ. *J. of Logic and Computation*, 17(6):1109–1134, 2007. doi:10.1093/logcom/exm037.

12 José Espírito Santo. Completing herbelin's programme. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *LNCS*, pages 118–132. Springer, 2007. doi:10.1007/978-3-540-73228-0_10.

13 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993. doi:10.1145/155090.155113.

14 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011. doi:10.1016/j.ic.2010.09.004.

15 Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969. doi:10.1007/BF01201353.

16 Ulysse Gérard and Dale Miller. Separating functional computation from relations. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *LIPIcs*, pages 23:1–23:17, 2017. doi:10.4230/LIPIcs.CSL.2017.23.

17 Jean-Yves Girard. A new constructive logic: classical logic. *Math. Structures in Comp. Science*, 1:255–296, 1991. doi:10.1017/S0960129500001328.

18 Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, June 2001. doi:10.1017/S096012950100336X.

19 Clemens Grabmayer. Modeling terms by graphs with structure constraints (two illustrations). In Maribel Fernández and Ian Mackie, editors, *TERMGRAPH@FSCD*, volume 288 of *EPTCS*, pages 1–13, 2018. doi:10.48550/arXiv.1902.02010.

20 Stéphane Graham-Lengrand. Polarities and focussing: a journey from realisability to automated reasoning, December 2014. Habilitation à diriger des recherches. URL: https://tel.archives-ouvertes.fr/tel-01094980.

21 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

22 Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Computer Science Logic, 8th International Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1995. doi:10.1007/BFb0022247.

23 Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. PhD thesis, Université Paris 7, 1995. URL: https://tel.archives-ouvertes.fr/tel-00382528.

24 Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi. doi:10.1016/j.tcs.2009.07.041.

25 Chuck Liang and Dale Miller. A focused approach to combining logics. *Annals of Pure and Applied Logic*, 162(9):679–697, 2011. doi:10.1016/j.apal.2011.01.012.

26 Chuck Liang and Dale Miller. Focusing Gentzen's LK proof system. In Thomas Piecha and Kai Wehmeier, editors, *Peter Schroeder-Heister on Proof-Theoretic Semantics*, Outstanding Contributions to Logic. Springer, 2022. To appear. URL: https://hal.archives-ouvertes.fr/hal-03457379.

27 Sonia Marin, Dale Miller, Elaine Pimentel, and Marco Volpe. From axioms to synthetic inference rules via focusing. *Annals of Pure and Applied Logic*, 173(5):1–32, 2022. doi:10.1016/j.apal.2022.103091.

28 Dale Miller. Mechanized metatheory revisited. *Journal of Automated Reasoning*, 63(3):625–665, October 2019. doi:10.1007/s10817-018-9483-3.

29 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012. doi:10.1017/CBO9781139021326.

30 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1–2):125–157, 1991. doi:10.1016/0168-0072(91)90068-W.

31 Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005. doi:10.1145/1094622.1094628.

32 Dale Miller and Jui-Hsuan Wu. A positive perspective on term representations: Extended paper. Technical report, Inria, 2022. URL: https://hal.inria.fr/hal-03843587.

33 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, 100(1):41–77, 1992. doi:10.1016/0890-5401(92)90009-5.

34 Guillaume Munch-Maccagnoni and Gabriel Scherer. Polarised intermediate representation of lambda calculus with sums. In *30th Symp. on Logic in Computer Science*, pages 127–140. IEEE Computer Society, 2015. doi:10.1109/LICS.2015.22.

35 Elaine Pimentel, Vivek Nigam, and João Neto. Multi-focused proofs with different polarity assignments. In Mario Benevides and Rene Thiemann, editors, *Proc. of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, volume 323 of *ENTCS*, pages 163–179, July 2016. doi:10.1016/j.entcs.2016.06.011.

36 Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001. doi:10.1007/s001530100091.

37 Robert J. Simmons. Structural focalization. *ACM Trans. on Computational Logic*, 15(3):21, 2014. doi:10.1145/2629678.

38 Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2):13:1–13:35, 2010. doi:10.1145/1656242.1656248.

39 Philip Wadler. Call-by-value is dual to call-by-name. In *8th Int. Conf. on Functional Programming*, pages 189–201, New York, NY, 2003. ACM. doi:10.1145/944705.944723.

40 Keven Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: The propositional fragment. In *Post-proceedings of TYPES 2003 Workshop*, number 3085 in LNCS. Springer, 2003. doi:10.1007/978-3-540-24849-1_23.

41 Noam Zeilberger. Focusing and higher-order abstract syntax. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 359–369. ACM, 2008. doi:10.1145/1328897.1328482.

42 Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1), 2008. Special issue on classical logic and computation. doi:10.1016/j.apal.2008.01.001.