A positive perspective on term representation: work in progress

Dale Miller
Inria Saclay & LIX, Institut Polytechnique de Paris

Jui-Hsuan Wu
LIX, Institut Polytechnique de Paris

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 bias, *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 bias, *LJF* proofs yield a structure in which explicit sharing of term structures is possible in the style of administrative normal form. In this situation, 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 techniques from concurrency theory—namely traces and simulation—to compare untyped λ-terms using such different structuring disciplines.

1 Introduction

The structure of terms and expressions are represented variously via, say, 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. 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* [34]) seems to be too restrictive, and sequent calculus seems to have too little structure. Instead, we use the *LJF* proof system [23], 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 [32, 37, 40]. Instead of dealing with functional programming style evaluation, we address a more primitive notion: what is a *term* (which is used to encode programs) and what is *substitution* (which is used to describe various kinds of evaluation). Even this more narrow setting has been addressed using the focused proof systems *LJT* [12, 21] and *LJQ* [11]. As in those papers, we shall consider proofs-as-terms and not proofs-as-programs since we do not consider evaluation of such structures in this paper.

Submitted to:
LFMTP 2022

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 [29]. 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 both intuitionistic and classical logic as the *LJF* and *LKF* proof systems [23]. This paper considers only intuitionistic logic formula built with 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*. Occurrences of logical connectives introduced by the left and right introduction rules only appear in the non-storage zones Θ and Δ. Sequents of the form $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow \Delta$ are *border sequents*: when we define synthetic inference rules in Definition 3, these sequents form the borders (the conclusion and premises) of synthetic inference rules.

Notational conventions: We shall 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 implication is unambiguous (they are *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.

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 it is 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 positive, N is negative, A is an atomic formula, and B, B', and C denote arbitrary formulas. Note that in our setting, if P is positive, it 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

Decide, Release, and Store Rules

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

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

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* in which the only introduction rules are for implications.

the decide rules in that release rules sit on top of a $\Downarrow$-phase while their premises are at the bottom of an $\Uparrow$-phase.

The key idea behind focused systems is the separation of those inference rules that are invertible and those that are non-invertible. Inference rules in the $\cdot \Uparrow$-phase are invertible and do not need require information external to these sequents. On the other hand, rules with $\cdot \Downarrow$-phase may not be invertible and a complete search strategy may need to backtrack over choices made by those inference rules. The decide (D_l and D_r) and release (R_l and R_r) rules are responsible for switching between these two phases.

The following soundness and completeness theorem for *LJF* can be found in [23], 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 1. 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 present the notion of the *order of a formula* and show two technical results about order of formulas within *LJF* proofs.

Definition 2 (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 1. Let B be a formula such that $ord(B) \le n$. 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 straightforward induction on the structure of cut-free proofs.

Proposition 2. Let Ξ be a cut-free *LJF* proof of $\Gamma \Uparrow \cdot \vdash \cdot \Uparrow A$ 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 a similar definition in [26].

Definition 3 (Synthetic inference rule)**.** A *left synthetic inference rule* is an inference rule of the form

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

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 4 (Bipole for B)**.** Let B be a negative polarized formula in *LJF*. A *bipole for B* is a synthetic inference rule for B 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 an example.

Definition 5 (Rules from polarized theory)**.** Let $\mathcal{T}$ be a finite set of formulas of order 2 or less, and let δ be an atomic bias assignment. We define $LJ\langle\delta,\mathcal{T}\rangle$ to be the extension of *LJ* with the left synthetic inference rules that arise from the formulas in the (δ-polarized) theory $\mathcal{T}$. More precisely, for every left synthetic inference rule

$$\dfrac{\mathcal{T},\Gamma_1 \vdash \Delta_1 \quad \dots \quad \mathcal{T},\Gamma_n \vdash \Delta_n}{\mathcal{T},\Gamma \vdash \Delta}\; B, \qquad \text{where } B \in \mathcal{T}, \text{ then the inference rule} \qquad \dfrac{\Gamma_1 \vdash \Delta_1 \quad \dots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\; B$$

is added to $LJ\langle\delta,\mathcal{T}\rangle$. The theory $\mathcal{T}$ is not present as formulas in the sequents of $LJ\langle\delta,\mathcal{T}\rangle$ but as inference rules.

Example 1. Let $a_0, a_1, \dots$ be a sequence of atomic (propositional) formulas. Let $\mathcal{T}$ be the collection of the following Horn clauses: D_1 is $a_0 \supset a_1$, D_2 is $a_0 \supset a_1 \supset a_2$, and more generally,

$$D_n = a_0 \supset \cdots \supset a_{n-1} \supset a_n \quad (n > 0).$$

Let $\delta^-(\cdot)$ be the bias assignment that gives all atomic formulas the negative bias and let $\delta^+(\cdot)$ be the bias assignment that gives all atomic formulas the positive bias. The inference rules in $LJ\langle\delta^-,\mathcal{T}\rangle$ include

$$\dfrac{\Gamma \vdash a_0}{\Gamma \vdash a_1} \qquad \dfrac{\Gamma \vdash a_0 \quad \Gamma \vdash a_1}{\Gamma \vdash a_2} \qquad \dots \qquad \dfrac{\Gamma \vdash a_0 \quad \cdots \quad \Gamma \vdash a_{n-1}}{\Gamma \vdash a_n} \qquad \dots$$

Given these inference rules, there is a *unique* proof of $a_0 \vdash a_n$ and that proof has 2^n occurrences of these inference rules and initial rules. The negative bias assignment to atomic formulas yields the *backchaining* interpretation of these Horn clauses. In contrast, the inference rules in $LJ\langle\delta^+,\mathcal{T}\rangle$ are

$$\frac{\Gamma, a_0, a_1 \vdash A}{\Gamma, a_0 \vdash A} \qquad \frac{\Gamma, a_0, a_1, a_2 \vdash A}{\Gamma, a_0, a_1 \vdash A} \qquad \cdots \qquad \frac{\Gamma, a_0, \ldots, a_{n-1}, a_n \vdash A}{\Gamma, a_0, \ldots, a_{n-1} \vdash A} \qquad \cdots$$

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 Horn clauses.

There are a number of proofs in the literature that show how cut can be eliminated within focusing proofs. Some such proofs, such as the one used by Liang & Miller in [23, 25], introduce different variants of the cut rule and follow a rather tedious argument detailing how these cut rule moves through individual inference rules within $\Uparrow$ and $\Downarrow$ 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 can move through entire phases at a time. Other phase-based cut-elimination proofs appear in [6, 24, 33, 39, 40]. Part II of Graham-Lengrand's HdR dissertation [20] and Simmons's article [35] provide overviews of such cut-elimination proofs. None of the proofs mentioned above deal directly with *synthetic* rules in the sense that we have defined them here. In particular, synthetic rules are not identified with phases and they do not come in either negative or positive variants. The following theorem, which deals with cut-elimination in the presence of synthetic rules, is proved in [26].

Theorem 2 (Cut admissibility for $LJ\langle\delta,\mathcal{T}\rangle$)**.** Let $\mathcal{T}$ be a set of formulas of order 2 or less and let $\delta(\cdot)$ be an atomic bias assignment. The cut rule is admissible for the proof system $LJ\langle\delta,\mathcal{T}\rangle$.

If we limit our theory $\mathcal{T}$ to contain formulas of order 2 or less and if we are only interested in proving sequents of the form $A_1,\ldots,A_n \vdash A_0$ (for atomic $A_0,\ldots,A_n$) then the only inference rules from $LJ\langle\delta,\mathcal{T}\rangle$ that we need are those that correspond to synthetic rules and the weakening, contraction, and initial rules.

3 A term calculus for *LJF*

A term calculus for *LJF*, called the $\lambda\kappa$-calculus, was introduced by Brock-Nannestad et al. in [3] in order to study how cut-elimination and polarization can be used to describe different evaluation strategies (although we shall not consider its uses with evaluation). The portion of their calculus that corresponds to cut-free *LJF* proofs contains three syntactic categories and seven constructors and is displayed in Figure 2. As is clear from the λProlog [28] specification of the syntax of $\lambda\kappa$-calculus terms in Figure 2, we are making use of meta-level binding to capture $\lambda\kappa$-calculus bindings. The expression $\kappa(\lambda x.t)$ is abbreviated as $\kappa x.t$. The term constructor written using the boldface $\boldsymbol{\lambda}$ is of type $(val \to tm) \to tm$: the expression $\boldsymbol{\lambda} x.t$ is encoded as `(abs x\t)` in λProlog[1]. At the same time, we are using meta-level application to capture the application of constructors: for example, the term constructor written as $x \frown k$ is encoded using two meta-level applications `((app x) k)`.

Figure 3 contains an annotated version of the *LJF* proof system. Since the order in which $\Uparrow$ rules are applied is not important, we have merged the R_l and S_l rules into one rule and we have restricted the S_r rule so that it is applied only when the left non-storage zone is empty.

[1]In λProlog, λ-abstraction is written using an infix backslash.

$$\begin{array}{rrl} \textit{Terms}: & t,u ::= & \lambda x.t \mid x^\frown k \mid \lceil p \rceil \\ \textit{Values}: & p,q ::= & x \mid \lfloor t \rfloor \\ \textit{Continuations}: & k ::= & \epsilon \mid p :: k \mid \kappa x.t \end{array}$$

```
kind tm, val, cont          type.

type abs       (val -> tm) -> tm.
type app       val -> cont -> tm.
type cl                val -> tm.
type fl                tm -> val.
type epsilon                cont.
type cns     val -> cont -> cont.
type kappa   (val -> tm) -> cont.
```

Figure 2: Declarations for the $\lambda\kappa$-calculus along with the corresponding λProlog specification.

Decide, Release, and Store Rules

$$\frac{x : N, \Gamma \Downarrow N \vdash k : A}{x : N, \Gamma \vdash x^\frown k : A}\ D_l \qquad \frac{\Gamma \vdash p : P \Downarrow}{\Gamma \vdash \lceil p \rceil : P}\ D_r \qquad \frac{\Gamma \vdash t : N \Uparrow}{\Gamma \vdash \lfloor t \rfloor : N \Downarrow}\ R_r$$

$$\frac{\Gamma, x : P \vdash t : A}{\Gamma \Downarrow P \vdash \kappa x.t : A}\ R_l/S_l \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : A \Uparrow}\ S_r$$

Initial Rules

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

Introduction Rules for implication

$$\frac{\Gamma \vdash p : B \Downarrow \qquad \Gamma \Downarrow B' \vdash k : A}{\Gamma \Downarrow B \supset B' \vdash p :: k : A}\ \supset L \qquad \frac{\Gamma, x : B \vdash t : B' \Uparrow}{\Gamma \vdash \lambda x.t : B \supset B' \Uparrow}\ \supset R/S_l$$

Figure 3: $\lambda\kappa$-terms annotating some *LJF* inference rules

Given that synthetic inference rules are collections of *LJF* inference rules, synthetic inference rules can be annotated by collections of $\lambda\kappa$-calculus constructors, which we shall refer to here as *combinators*. The syntactic types of combinators will contain *val* and *tm* but not *cont*: this latter type is used only *internal* to the construction of synthetic inference rules.

N.B.: Formally, we encode syntactic categories as type expressions (built using primitive types and the arrow constructor). From the point of view of *LJF*, however, type expressions are formally encoded as propositional formulas (built using atomic formulas and implications). Thus, we find it convenient to conflate the notions of syntactic categories and types, as well as primitive types and atomic formulas.

Example 2. Consider annotating the theory in Example 1 as follows:

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

If all atomic formulas are biased negatively, then the annotated synthetic rules are of the form

$$\frac{\Gamma \vdash t_0 : a_0 \quad \cdots \quad \Gamma \vdash t_{n-1} : a_{n-1}}{\Gamma \vdash (E_n\ t_0\ \cdots t_{n-1}) : a_n}\ ,$$

where $n \geq 1$ and the combinators corresponding to these rules are given by

$$E_1 = \lambda x_0.\, d_1{}^\frown(\lfloor x_0 \rfloor :: \epsilon), \quad \ldots, \quad E_n = \lambda x_0 \ldots \lambda x_{n-1}.\, d_n{}^\frown(\lfloor x_0 \rfloor :: \cdots :: \lfloor x_{n-1} \rfloor :: \epsilon), \ \ldots .$$

$$\begin{array}{l} (E_4\ (E_3\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))) \\ \quad (E_3\ (E_2\ (E_1\ E_0)\ (E_1\ E_0))\ (E_2\ (E_1\ E_0)\ (E_1\ E_0)))) \end{array} \qquad \begin{array}{ll} (F_1\ d_0 & (\lambda x_1. \\ (F_2\ d_0\ x_1 & (\lambda x_2. \\ (F_3\ d_0\ x_1\ x_2 & (\lambda x_3. \\ (F_4\ d_0\ x_1\ x_2\ x_3 & (\lambda x_4. \lceil x_4 \rceil)))))))) \end{array}$$

Figure 4: Two proofs of a_4 from the context $\{d_0 : a_0\}$, where $E_0 = d_0{}^\frown\epsilon$.

The syntactic type of E_n $(n \geq 1)$ is $tm \to \cdots \to tm \to tm$ (where tm occurs $n+1$ times). On the other hand, if all atomic formulas are biased positively, then the annotated synthetic rules are

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

where $n \geq 1$ and the combinators corresponding to these rules are given by

$$F_1 = \lambda x_0 \lambda k.\ d_1{}^\frown(x_0 :: \kappa k), \quad \ldots, \quad F_n = \lambda x_0 \ldots \lambda x_{n-1} \lambda k.\ d_n{}^\frown(x_0 :: \cdots :: x_{n-1} :: \kappa k),\ \ldots\ .$$

Since the expression $\kappa x.t$ is an abbreviation for $\kappa(\lambda x.t)$, when we write above $\cdots \lambda k. \cdots (\kappa k) \cdots$, the bound variable k has syntactic type $val \to tm$. The syntactic type of F_n $(n \geq 1)$ is $val \to \cdots \to val \to (val \to tm) \to tm$ (where val occurs $n+1$ times). Figure 4 displays the unique proof of a_4 using the E_n combinators and the shortest proof of a_4 using the F_n combinators: the structure on the right allows for explicit sharing of subterms while the structure on the left must repeat these subterms.

4 Terms over a first-order signature

Let i be a primitive type and let Σ be the first-order signature $\{z\colon i,\ s\colon i \to i,\ f\colon i \to i \to i\}$. By the expression "$\Sigma$-terms of type i" we usually mean terms such as $(s\ z)$, $(s\ (s(s\ z)))$, $(f\ (f\ z\ z)\ (s\ z))$, etc. In order to model terms using this signature in $\lambda\kappa$-calculus, we consider instead the signature $\{z\colon val, s\colon val, f\colon val\}$ and consider terms t that annotate *LJF* proofs of the endsequent

$$z\colon i,\ s\colon i \supset i,\ f\colon i \supset i \supset i \Uparrow \cdot \vdash \cdot \Uparrow t\colon i.$$

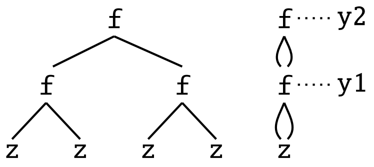
The structure of terms such as t depends on the polarity of i. When i is given a negative bias, we have terms such as

$$f{}^\frown(\lfloor f{}^\frown(\lfloor z{}^\frown\epsilon\rfloor :: \lfloor z{}^\frown\epsilon\rfloor :: \epsilon)\rfloor :: \lfloor f{}^\frown(\lfloor z{}^\frown\epsilon\rfloor :: \lfloor z{}^\frown\epsilon\rfloor :: \epsilon)\rfloor :: \epsilon),$$

which is more commonly abbreviated as simply $f\ (f\ z\ z)\ (f\ z\ z))$. When i is given a positive bias, we have terms such as $f{}^\frown(z :: z :: \kappa y_1.(f{}^\frown(y_1 :: y_1 :: \kappa y_2.\lceil y_2\rceil)))$. In order to make expressions such as this more readable, we introduce a notation using the `name` keyword, which, for this example, is written as

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

These two terms can be viewed as the following labeled tree and DAG.



In general, we shall use the `name` syntax to abbreviate $\lambda\kappa$-term terms of the form $f^\frown(v_1 :: \ldots :: v_n :: \kappa y.t)$ (where $n > 0$) as the expression "`name` $y = (f\ v_1 \cdots\ v_n)$ `in` t." In both of these expressions, the bound variable y has scope over t.

4.1 Intermediate representation of programs

A useful intermediate representation of programs in compilers of functional programming language 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 biased syntax, the expressions that result are in ANF. Usually, the keyword `let` is used instead of `name` in the presentation of such syntax. We have made this change deliberately since the proof theory instructs us that we always link a `name` with an actual application of a function symbol. This linkage is not customarily assumed when one uses the `let` keyword: for example, we can write

```
let x = z in let y = z in (f x y).
```

The closest we can come to this in our positive bias assignment syntax is the expression

```
name w = (f z z) in w.
```

We are not able to write "`name x = z in`" since `z` is not applied to any arguments.

Given an expression E in negative bias assignment, 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 operations are often called *common subexpression elimination*. The positive bias assignment 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.

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, 22, 35]. In all of these references, however, atomic formulas are given the negative bias.

4.2 Encoding functional expressions as relational queries

Programmers of Prolog often have the following issue. Mathematical expressions, such as $\sqrt{b^2 - 4ac}$, are calls to various functions (here, subtraction, addition, multiplication, and square root). Logic programming is, however, based on relations: for example, addition on real numbers can be 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 of these primitive functions. One way to organize these relations to compute the expression above involves converting that expression into ANF, such as

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

As described in [16], it is straightforward to convert such an ANF expression into a series of calls to predicates. In particular, we can rewrite this expression by replacing the **name** $n = f\ x_1\ \cdots\ x_i$ **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 in Prolog.

5 The untyped λ-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 the sense 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 $\Gamma_0 \subseteq \Gamma$).

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

These derivations can be extended to have border sequents as premises only 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

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

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

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

Without annotations, these inference rules are not illuminating. However, we can annotate *LJF* proofs with sequents of the form $\Psi : (D \to D) \to D,\ \Phi : D \to (D \to D),\ x_1 : D,\ \ldots,\ x_n : D \vdash t : D$, where Ψ and Φ are two $\lambda\kappa$-calculus variables of type *val*. In this case, the term t represents an untyped λ-term with free variables among $x_1,\ \ldots,\ x_n$. As we shall see next, there are two different term representations for untyped λ-term based on annotating *LJF* proofs.

N.B.: There are two notions of typing occurring here. The symbol, say, Ψ has a type within the $\lambda\kappa$-calculus setting of *val* whereas in our annotated sequent setting, Ψ is also *associated* with the formula $D \supset (D \supset D)$: for convenience, that association is written using the familiar syntax of typing.

5.1 Using negative-bias syntax for the untyped λ-calculus

If we polarize D negatively, we get the standard tree-like representation for the untyped λ-calculus. In particular, consider an annotated *LJF* proof of a sequent of the form $\Gamma_0, x_1 : D, \ldots, x_n : D\Uparrow \cdot \vdash \cdot \Uparrow t : D$. The synthetic inference rules in such a proof are either the result of deciding on $x_i : D$ (for $1 \le i \le n$) followed by I_l or on one of the two formulas in Γ_0. Figure 5 supplies the details for these three synthetic rules. These synthetic rules yield the following three combinators: these are listed with their λProlog name, a syntactic type, and a definition as $\lambda\kappa$-terms.

`type napp  tm -> tm -> tm.`	$\lambda t \lambda s.\Phi ^\frown (\lfloor t \rfloor :: \lfloor s \rfloor :: \epsilon)$
`type nabs  (val -> tm) -> tm.`	$\lambda r.\Psi ^\frown (\lfloor \lambda r \rfloor :: \epsilon)$
`type nvar  val -> tm.`	$\lambda x.x ^\frown \epsilon$

The untyped λ-term $((\lambda x.xx)(\lambda x.xx))$ can be written as the following term of type *tm*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : D \vdash t : D}{\Gamma, x : D \vdash t : D\Uparrow}\,S_r}{\Gamma \vdash \lambda x.t : D \supset D \Uparrow}\,\supset R/S_l}{\Gamma \vdash \lfloor \lambda x.t \rfloor : D \supset D \Downarrow}\,R_r \qquad \dfrac{}{\Gamma \Downarrow D \vdash \epsilon : D}\,I_l}{\dfrac{\Gamma \Downarrow (D \supset D) \supset D \vdash \lfloor \lambda x.t \rfloor :: \epsilon : D}{\Gamma \vdash \Psi^\frown(\lfloor \lambda x.t \rfloor :: \epsilon) : D}\,D_l}\,\supset L$$

$$\dfrac{\dfrac{\Gamma \vdash t : D}{\Gamma \vdash \lfloor t \rfloor : D \Downarrow}\,R_r \qquad \dfrac{\Gamma \vdash s : D}{\Gamma \vdash \lfloor s \rfloor : D \Downarrow}\,R_r \qquad \dfrac{}{\Gamma \Downarrow D \vdash \epsilon : D}\,I_l}{\dfrac{\Gamma \Downarrow D \supset D \supset D \vdash \lfloor t \rfloor :: \lfloor s \rfloor :: \epsilon : D}{\Gamma \vdash \Phi^\frown(\lfloor t \rfloor :: \lfloor s \rfloor :: \epsilon) : D}\,D_l}\,\supset L\times 2$$

$$\dfrac{\dfrac{}{\Gamma \Downarrow D \vdash \epsilon : D}\,I_l}{\Gamma \vdash x_i{}^\frown \epsilon : D}\,D_l$$

Figure 5: The three derivations that justify the three kinds of synthetic rules when D has negative bias.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : D \vdash t : D}{\Gamma, x : D \vdash t : D\Uparrow}\,S_r}{\Gamma \vdash \lambda x.t : D \supset D \Uparrow}\,\supset R/S_l}{\Gamma \vdash \lfloor \lambda x.t \rfloor : D \supset D \Downarrow}\,R_r \qquad \dfrac{\Gamma, y : D \vdash s : D}{\Gamma \Downarrow D \vdash \kappa y.s : D}\,R_l/S_l}{\dfrac{\Gamma \Downarrow (D \supset D) \supset D \vdash \lfloor \lambda x.t \rfloor :: \kappa y.s : D}{\Gamma \vdash \Psi^\frown(\lfloor \lambda x.t \rfloor :: \kappa y.s) : D}\,D_l}\,\supset L$$

$$\dfrac{\dfrac{}{\Gamma \vdash x_i : D \Downarrow}\,I_l \qquad \dfrac{}{\Gamma \vdash x_j : D \Downarrow}\,I_l \qquad \dfrac{\Gamma, y : D \vdash t : D}{\Gamma \Downarrow D \vdash \kappa y.t : D}\,R_l/S_l}{\dfrac{\Gamma \Downarrow D \supset D \supset D \vdash x_i :: x_j :: \kappa y.t : D}{\Gamma \vdash \Phi^\frown(x_i :: x_j :: \kappa y.t) : D}\,D_l}\,\supset L\times 2$$

$$\dfrac{\dfrac{}{\Gamma \vdash x_i : D \Downarrow}\,I_r}{\Gamma \vdash \lceil x_i \rceil : D}\,D_r$$

Figure 6: The three derivations that justify the three kinds of synthetic rules when D has positive bias.

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

The prefix `n` on these names is meant to remind us that we assigned D the negative bias. Systems, such as λProlog, Abella, LF, and Isabelle, typically encode the untyped λ-calculus in this fashion.

5.2 Using positive-bias syntax for the untyped lambda-calculus

If we polarize D positively, we get a different format for 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. As a result, we have the following three combinators (see Figure 6).

`type papp  val -> val -> (val -> tm) -> tm.`	$\lambda u \lambda v \lambda k.\Phi^\frown(u :: v :: \kappa k)$
`type pabs  (val -> tm) -> (val -> tm) -> tm.`	$\lambda r \lambda k.\Psi^\frown(\lfloor \lambda r \rfloor :: \kappa k)$
`type pvar  val -> tm.`	$\lambda x.\lceil x \rceil$

Using these combinators, 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`. The prefix `p` on these names is meant to remind us that, in this case, we assigned D the positive bias.

As before, we use `name` expressions to present syntax using positive bias. In particular, the expression `(papp u v (w\ Body))` can be seen as first building an application of the variables `u` and `v` and then *naming* that application as `w` in the scope of `Body`. An alternative syntax could be `name w = (app u v) in Body`. Similarly, the expression `(pabs R (w\ Body))` can be seen as first building an abstraction from `R` then naming that abstraction as `w` in the scope of `Body`. An alternative syntax could be `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 w\ pvar w)))
```

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 w = (app z u) in w.
```

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 w  = (app z1 u) in w

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 w  = (app z u2) in w
```

The first of these terms illustrates that a named structures 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 rather 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.

5.3 Tracing untyped λ-terms

Given that we have two very 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 5.2 (based on positive bias assignment) as equivalent in some sense to each other and to the expression (based on negative bias assignment)

```
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 7: Traces through negative and positive-biased expressions.

```
(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 [33] where proofs based on positive bias assignment to atomic formulas are systematically converted to proofs based on negative bias assignment.

We propose instead to use a "black box" approach in which we probe a term in order to describe *traces* within expressions. For example, we can ask whether or not the term denotes a top-level application or not. This is easy to check for negative-biased expressions by simply checking 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.

```
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 7. Note that the order 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

[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 for such a specification.

```
type w       val.
type wtrace  trace.

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

We shall say that two expressions denoting untyped λ-terms (using either positive or negative bias assignment) are *trace equivalent* if they both have the same set of 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-biased expressions: in particular, the examples at the end of Section 5.2 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 (W1\ left (bnd (W2\ left W1)))),
                      (bnd (W1\ left (bnd (W2\ right W2)))),
                      (bnd (W1\ right W1))].
T = nabs (W1\ napp (nabs (W2\ napp (nvar W1) (nvar W2))) (nvar W1))
```

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 [28, Section 7.4.2]).

5.4 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. The paper [9] develops a graphic representation of sharing in the untyped λ-calculus using *λgraphs*. 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-biased syntax effectively, we use 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.

Note that the top-level of an untyped λ-term expression based on a 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. This presentation suggests the following Abella declarations (which are similar to λProlog declarations).

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

[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.

```
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 P', faction P A P' -> exists Q', faction Q A Q' /\ sim Vs P' Q') /\
   (forall P',   baction P P'   -> exists Q', baction Q Q'   /\
                                     nabla u, sim (u::Vs) (P' u) (Q' u)).
```

Figure 8: An Abella specification of sharing simulation

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

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

For example, the last term displayed in Section 5.2 can be written as the Abella term

```
(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.) Positive-biased expressions 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 given labels. These labels, called *actions* here, are of the following three kinds (see the full specification in Figure 8).

1. Primitive actions are described using the predicate `paction`. Such actions simply 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, 30] (written in Abella as `nabla`) is used to manage nominal constants and binding mobility [27]. 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 [31]. Our specification of simulation in the presence of both free and bound actions follows the specification technique of Tiu & Miller [36] that used the ∇-quantifier. Given our specification of simulation, the specification of bisimulation is easy to write as well.

6 Cut elimination at the level of synthetic inference rules

Theorem 2 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 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 bias 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_1 and Ξ_2 are cut-free.

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \vdash u : E'\end{array} \qquad \begin{array}{c}\Xi_2\\ \Gamma, x : E' \vdash t : E\end{array}}{\Gamma \vdash Cut_0(x.t, u) : E}\ Cut_0$$

While the term $Cut_0(x.t,u)$ is not a term, it names 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.

A detailed presentation of the cut-elimination procedure is given in Appendix B. Here we illustrate how this procedure works on our two encodings of untyped λ-terms. In particular, we will provide specifications (using λProlog code) to define the predicates

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

that have the following specification: given `T` of type `tm` and `R` of type `val -> tm` then `(nsubst T R S)` is provable if and only if `T`, `R`, and `S` use the combinators `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 `T`, `R`, and `S` built using the combinators `papp`, `pabs`, and `pvar`.

When terms are built using the negative bias, 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 combinators recursively. We have thus:

```
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 here substitution moves recursively through the second (abstracted) argument in order to compute the substitution. This style of substitution is, of course, the familiar one. We can write this substitution operation as a postfix operator using the following functional equations.

- $(\texttt{nvar}\ x)[x/t] = t$;
- $(\texttt{nvar}\ y)[x/t] = (\texttt{nvar}\ y)$, 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 that y is not free in t.

When terms are built using the positive bias, 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 y1\ papp y1 y1 y2\ papp y2 y2 y3\ pvar y3)
          (x\ papp x x pvar) R.
R = papp w w (y1\ papp y1 y1 (y2\ papp y2 y2 (y3\ papp y3 y3 u\ pvar u)))
```

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 scoped 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 scoped of a binding on u.

Given our discussion of checking the simulation of two untyped λ-terms in Section 5.4, 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 `val`) 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 $\langle\Gamma \sqcup (\Gamma'[x := u]),\ u'[x := u]\rangle$, where $\sqcup$ denotes appending of two lists.

7 Related and future work

One goal in developing a logical framework, as we have done here with the *LJF* proof system and the $\lambda\kappa$-calculus, is to account for possibly many other calculi within that framework. Clearly, there have been many previous studies of term representation already in the literature, some of which have also been motivated by focused proof systems, such as [11, 12, 22]. Since *LJF* can easily account for several other focusing proof systems for intuitionistic logic [23], 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 limited to second order) would need to be generalized. In such a setting with higher-order constructors, cut-elimination should be able to derive and generalize *hereditary substitutions* [38].

As we mentioned in Section 5.4, the λgraphs in [9] are used to 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 [33].

The black box methods of analyzing the structure of terms with sharing (see Sections 5.3 and 5.4) 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]. In general, simulations can be seen as winning strategies, and there are known connections between winning strategies and focused proofs: see, for example, [10].

Acknowledgments: We thank Beniamino Accattoli and Kaustuv Chaudhuri for valuable discussions and suggestions. We also thank the reviewers for LFMTP 2022 for their comments on an earlier draft of this paper.

References

[1] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic*. *J. of Logic and Computation* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[2] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *Journal of Formalized Reasoning* 7(2), pp. 1–89, doi:10.6092/issn.1972-5787/4650.

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

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

[5] Iliano Cervesato & Frank Pfenning (2003): *A Linear Spine Calculus*. *Journal of Logic and Computation* 13(5), pp. 639–688, doi:10.1093/logcom/13.5.639.

[6] Kaustuv Chaudhuri (2008): *Focusing Strategies in the Sequent Calculus of Synthetic Connectives*. In Iliano Cervesato, Helmut Veith & Andrei Voronkov, editors: *LPAR: International Conference on Logic, Programming, Artificial Intelligence and Reasoning*, *LNCS* 5330, Springer, pp. 467–481, doi:10.1007/978-3-540-89439-1_33.

[7] Kaustuv Chaudhuri, Stefan Hetzl & Dale Miller (2016): *A Multi-Focused Proof System Isomorphic to Expansion Proofs*. *J. of Logic and Computation* 26(2), pp. 577–603, doi:10.1093/logcom/exu030.

[8] Kaustuv Chaudhuri, Dale Miller & Alexis Saurin (2008): *Canonical Sequent Proofs via Multi-Focusing*. In G. Ausiello, J. Karhumäki, G. Mauri & L. Ong, editors: *Fifth International Conference on Theoretical Computer Science*, *IFIP* 273, Springer, pp. 383–396, doi:10.1007/978-0-387-09680-3_26.

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

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

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

[12] José Espírito Santo (2007): *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*, *LNCS* 4583, Springer, pp. 118–132, doi:10.1007/978-3-540-73228-0_10.

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

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

[15] Gerhard Gentzen (1935): *Investigations into Logical Deduction*. In M. E. Szabo, editor: *The Collected Papers of Gerhard Gentzen*, North-Holland, Amsterdam, pp. 68–131, doi:10.1007/BF01201353. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969.

[16] Ulysse Gérard & Dale Miller (2017): *Separating Functional Computation from Relations*. In Valentin Goranko & Mads Dam, editors: *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, *LIPIcs* 82, pp. 23:1–23:17, doi:10.4230/LIPIcs.CSL.2017.23.

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

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

[19] Clemens Grabmayer (2018): *Modeling Terms by Graphs with Structure Constraints (Two Illustrations)*. In Maribel Fernández & Ian Mackie, editors: *TERMGRAPH@FSCD*, *EPTCS* 288, pp. 1–13, doi:10.48550/arXiv.1902.02010.

[20] Stéphane Graham-Lengrand (2014): *Polarities and Focussing: a journey from Realisability to Automated Reasoning*. Available at `https://tel.archives-ouvertes.fr/tel-01094980`. Habilitation à diriger des recherches.

[21] Hugo Herbelin (1995): *A Lambda-Calculus Structure Isomorphic to Gentzen-Style Sequent Calculus Structure*. In: *Computer Science Logic, 8th International Workshop, CSL '94*, *Lecture Notes in Computer Science* 933, Springer, pp. 61–75, doi:10.1007/BFb0022247.

[22] Hugo Herbelin (1995): *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. Ph.D. thesis, Université Paris 7. Available at `https://tel.archives-ouvertes.fr/tel-00382528`.

[23] Chuck Liang & Dale Miller (2009): *Focusing and Polarization in Linear, Intuitionistic, and Classical Logics*. *Theoretical Computer Science* 410(46), pp. 4747–4768, doi:10.1016/j.tcs.2009.07.041. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

[24] Chuck Liang & Dale Miller (2011): *A Focused Approach to Combining Logics*. *Annals of Pure and Applied Logic* 162(9), pp. 679–697, doi:10.1016/j.apal.2011.01.012.

[25] Chuck Liang & Dale Miller (2022): *Focusing Gentzen's LK proof system*. In Thomas Piecha & Kai Wehmeier, editors: *Peter Schroeder-Heister on Proof-Theoretic Semantics*, Outstanding Contributions to Logic, Springer. Available at `https://hal.archives-ouvertes.fr/hal-03457379`. To appear.

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

[27] Dale Miller (2019): *Mechanized Metatheory Revisited*. *Journal of Automated Reasoning* 63(3), pp. 625–665, doi:10.1007/s10817-018-9483-3.

[28] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press, doi:10.1017/CBO9781139021326.

[29] Dale Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform Proofs as a Foundation for Logic Programming*. *Annals of Pure and Applied Logic* 51(1–2), pp. 125–157, doi:10.1016/0168-0072(91)90068-W.

[30] Dale Miller & Alwen Tiu (2005): *A proof theory for generic judgments*. *ACM Trans. on Computational Logic* 6(4), pp. 749–783, doi:10.1145/1094622.1094628.

[31] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, Part II*. *Information and Computation* 100(1), pp. 41–77, doi:10.1016/0890-5401(92)90009-5.

[32] Guillaume Munch-Maccagnoni & Gabriel Scherer (2015): *Polarised Intermediate Representation of Lambda Calculus with Sums*. In: *30th Symp. on Logic in Computer Science*, IEEE Computer Society, pp. 127–140, doi:10.1109/LICS.2015.22.

[33] Elaine Pimentel, Vivek Nigam & João Neto (2016): *Multi-focused Proofs with Different Polarity Assignments*. In Mario Benevides & Rene Thiemann, editors: *Proc. of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, *ENTCS* 323, pp. 163–179, doi:10.1016/j.entcs.2016.06.011.

[34] Jan von Plato (2001): *Natural Deduction with General Elimination Rules*. *Archive for Mathematical Logic* 40(7), pp. 541–567, doi:10.1007/s001530100091.

[35] Robert J. Simmons (2014): *Structural Focalization*. *ACM Trans. on Computational Logic* 15(3), p. 21, doi:10.1145/2629678.

[36] Alwen Tiu & Dale Miller (2010): *Proof Search Specifications of Bisimulation and Modal Logics for the π-calculus*. *ACM Trans. on Computational Logic* 11(2), pp. 13:1–13:35, doi:10.1145/1656242.1656248.

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

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

[39] Noam Zeilberger (2008): *Focusing and higher-order abstract syntax*. In George C. Necula & 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*, ACM, pp. 359–369, doi:10.1145/1328897.1328482.

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

```
type pptrace               tm -> trace -> o.
type pppleft, pppright   val -> val -> o.
type pppabs              val -> (val -> tm) -> o.

pptrace (papp U V K) P :- pi v\ pppleft v U => pppright v V =>
                                                  pptrace (K v) P.
pptrace (pabs R K)   P :- pi v\ pppabs  v R => pptrace (K v) P.

pptrace (pvar V) (left  P) :- pppleft  V U, pptrace (pvar U) P.
pptrace (pvar V) (right P) :- pppright V U, pptrace (pvar U) P.
pptrace (pvar V) (bnd Q)   :- pppabs   V R,
              pi p\ pi u\ pptrace (pvar u) p => pptrace (R u) (Q p).
```

Figure 9: A second-order specification of `ptrace`

A A second-order specification of `ptrace`

The following λProlog specification of the predicate `pptrace` describes traces in expressions with positive bias assignment to D. Note that this specification involves clauses of only order two and that it contains three additional and auxiliary predicate constants.

B The cut-elimination procedure

$$\frac{\Gamma \vdash u : E' \qquad \Gamma, x : E' \vdash t : E}{\Gamma \vdash Cut_0(x.t, u) : E}\ Cut_0 \qquad \frac{\Gamma \vdash u : E \qquad \Gamma, x : E \vdash t : F \Uparrow}{\Gamma \vdash Cut_0(x.t, u) : F \Uparrow}\ Cut_0^*$$

$$\frac{\Gamma \vdash u : B \Uparrow \qquad \Gamma, x : B \vdash t : E}{\Gamma \vdash Cut_1(x.t, u) : E}\ Cut_1 \qquad \frac{\Gamma \vdash u : B \Uparrow \qquad \Gamma, x : B \vdash t : F \Uparrow}{\Gamma \vdash Cut_1(x.t, u) : F \Uparrow}\ Cut_1^*$$

Figure 10: Cut rules for LJF. Here, E and E' are atomic formulas, and B and F are arbitrary formulas.

In the following, we present a cut-elimination procedure for *LJF* which will be used to define the substitution no matter the bias assumption. The cut rules for *LJF* are presented in Figure 10.

First note that the elimination of Cut_0^* (resp. Cut_1^*) can be reduced to the case of Cut_0 (resp. Cut_1) by pushing the cut upwards through the $\Uparrow$-phase of the right premise. These two cut rules are applied only at the top-level and will not appear in the cut elimination procedure presented below.

Before presenting the cut-elimination procedure, we first give a theorem that justifies a "big-step" style presentation of such a procedure when cut rules are only applied to atomic formulas. In this case, only Cut_0 needs to be considered.

When D_l is used to focus on an implication on the left, above that occurrence of D_l must necessarily be a positive phase which will have above it zero or more negative phases before reaching border sequents. We call this collection of positive-below-negative-phases a *fused* phase.

Theorem 3. Let Π be a proof

$$\frac{\begin{array}{c}\Pi_1\\ \Gamma \vdash u : E'\end{array} \qquad \begin{array}{c}\Pi_2\\ \Gamma, x : E' \vdash t : E\end{array}}{\Gamma \vdash Cut_0(x.t, u) : E}\ Cut_0$$

with both Π_1 and Π_2 cut-free. We distinguish three cases:

- If Π_1 ends with D_r/I_r, then u is of the form $\lceil y \rceil$ with $y : E' \in \Gamma$. We can eliminate the cut by considering the proof

$$\begin{array}{c} \Pi_2[x \mapsto y] \\ \Gamma \vdash t[x \mapsto y] : E \end{array}$$

  where $\Pi_2[x \mapsto y]$ is obtained from Π_2 by replacing the variable x with y and by erasing the redundant occurrences of $y : E'$ in all the contexts.

- If Π_1 ends with D_l on $B_1 \supset \cdots \supset B_n \supset B$ with B a positive atom, then the cut moves to the left branch, above the fused phase, and becomes a cut between the last (rightmost) premise of the fused phase of Π_1 and the endsequent of Π_2.

- If Π_1 ends with D_l on $B_1 \supset \cdots \supset B_n \supset B$ with B a negative atom, then we have $E' = B$ and
  - if Π_2 ends with D_r/I_r, then t is of the form $\lceil y \rceil$ with $y : E \in \Gamma, x : E'$, then $y : E \in \Gamma$ and we can eliminate the cut by considering the proof

$$\cfrac{\cfrac{}{\Gamma \vdash y : E \Downarrow}\, I_r}{\Gamma \vdash \lceil y \rceil : E \Uparrow}\, D_r$$

  - if Π_2 ends with D_l on the cut formula E' (here we talk about the occurrence of E' in $x : E'$), then $E = E'$ and we can eliminate the cut by considering Π_1, and
  - if Π_2 ends with D_l on E'' with $y : E'' \in \Gamma$ for some y, then the cut moves to the right branch, above the fused phase in Π_2 and becomes a cut between the endsequent of Π_1 and each of the premises of the fused phase of Π_2.

This "big-step" cut-elimination procedure for *LJF* can be used to define a cut-elimination procedure for the extension $LJ\langle\delta,\Gamma_0\rangle$ of *LJ* with the polarized theory $\langle\delta,\Gamma_0\rangle$, which gives the definition of `nsubst` and `psubst` in Section 6.

We now present the full cut-elimination procedure.

Cut$_0$:

$$\cfrac{\begin{array}{c}\Pi_1 \\ \Gamma \vdash u : E'\end{array} \qquad \begin{array}{c}\Pi_2 \\ \Gamma, x : E' \vdash t : E\end{array}}{\Gamma \vdash Cut_0(x.t, u) : E}\, Cut_0$$

- if $\Pi_1 = \cfrac{\cfrac{}{\Gamma \vdash y : E' \Downarrow}\, I_r}{\Gamma \vdash u : E'}\, D_r$, then $u = \lceil y \rceil$ and $y : E' \in \Gamma$. We have thus $\begin{array}{c} \Pi_2[x \mapsto y] \\ \Gamma \vdash t[x \mapsto y] : E \end{array}$

- if $\Pi_1 = \cfrac{\begin{array}{c}\Xi_i \\ \Gamma \vdash p_i : B_i \Downarrow\end{array} \qquad \begin{array}{c}\Xi \\ \Gamma \Downarrow B \vdash k : E'\end{array}}{\Gamma \Downarrow B_1 \supset \cdots \supset B_n \supset B \vdash p_1 :: \cdots :: p_n :: k : E'}\, \supset L \times n$ with $n \geq 0$ and B atomic,

  then we have $u = w^\frown(p_1 :: \cdots :: p_n :: k)$ with $w : B_1 \supset \cdots \supset B_n \supset B \in \Gamma$.

If B is positive, then we have $\Xi = \dfrac{\overset{\Xi'}{\Gamma, z : B \vdash s : E'}}{\Gamma \Downarrow B \vdash k : E'}\ R_l/S_l$ with $k = \kappa z.s$. We have thus

$$\dfrac{\overset{\Xi_i}{\Gamma \vdash p_i : B_i \Downarrow} \qquad \dfrac{\dfrac{\overset{\Xi'}{\Gamma, z : B \vdash s : E'} \qquad \overset{\Pi_2^w}{\Gamma, z : B, x : E' \vdash t : E}}{\Gamma, z : B \vdash Cut_0(x.t, s) : E}\ Cut_0}{\Gamma \Downarrow B \vdash \kappa z.Cut_0(x.t, s) : E}\ R_l/S_l}{\dfrac{\Gamma \Downarrow B_1 \supset \cdots \supset B_n \supset B \vdash p_1 :: \cdots :: p_n :: \kappa z.Cut_0(x.t, s) : E}{\Gamma \vdash w\frown(p_1 :: \cdots :: p_n :: \kappa z.Cut_0(x.t, s)) : E}\ D_l}\ \supset L \times n$$

where Π_2^w is obtained from Π_2 by weakening.
If B is negative, then Ξ contains only the I_l rule and $E' = B$.

- if $\Pi_2 = \dfrac{\dfrac{}{\Gamma, x : E' \vdash y : E \Downarrow}\ I_r}{\Gamma, x : E' \vdash t : E}\ D_r$, then $t = \lceil y \rceil$ and we have $y : E \in \Gamma$ since $E \neq E'$ (they have different polarities). We have thus

$$\dfrac{\dfrac{}{\Gamma \vdash y : E \Downarrow}\ I_r}{\Gamma \vdash \lceil y \rceil : E}\ D_r$$

- if $\Pi_2 = \dfrac{\dfrac{\overset{\Lambda_j}{\Gamma' \vdash q_j : F_j \Downarrow} \qquad \overset{\Lambda}{\Gamma' \Downarrow F \vdash l : E}}{\Gamma' \Downarrow F_1 \supset \cdots \supset F_m \supset F \vdash q_1 :: \cdots :: q_m :: l : E}\ \supset L \times m}{\Gamma' \vdash t : E}\ D_l$ with $\Gamma' = \Gamma, x : E', m \geq 0$ and F atomic, then we have $t = z\frown(q_1 :: \cdots :: q_m :: l)$ with $z : F_1 \supset \cdots \supset F_m \supset F \in \Gamma'$.
If F_j is positive, then Λ_j contains only the I_r rule and $q_j : F_j \in \Gamma'$. Since $F_j \neq E'$, we have $q_j : F_j \in \Gamma$ and thus
$$\dfrac{}{\Gamma \vdash q'_j : F_j \Downarrow}\ I_r$$
with $q'_j = q_j$.
If F_j is negative, then Λ_j contains the R_r rule followed by an $\Uparrow$-phase. The border sequent obtained after this $\Uparrow$-phase is of the form $\Gamma', \Delta \vdash t_0 : E_0$. We have thus
$$\dfrac{\overset{\Pi_1^w}{\Gamma, \Delta \vdash u : E'} \qquad \Gamma', \Delta \vdash t_0 : E_0}{\Gamma, \Delta \vdash Cut_0(x.t_0, u) : E_0}\ Cut_0$$
where Π_1^w is obtained from Π_1 by weakening.
By reapplying the $\Uparrow$-phase and the release rule, we obtain
$$\overset{\Lambda'_j}{\Gamma \vdash q'_j : F_j \Downarrow}$$
for some q'_j.

If F is positive, then we have

$$\Lambda = \cfrac{\overset{\Lambda_0}{\Gamma', x' : F \vdash t' : E}}{\Gamma' \Downarrow F \vdash l : E}\ R_l/S_l$$

with $l = \kappa x'.t'$. Hence, we have

$$\Lambda' = \cfrac{\cfrac{\overset{\Pi_1^w}{\Gamma, x' : F \vdash u : E'} \qquad \overset{\Lambda_0}{\Gamma', x' : F \vdash t' : E}}{\Gamma, x' : F \vdash Cut_0(x.t', u) : E}\ Cut_0}{\Gamma \Downarrow F \vdash l' : E}\ R_l/S_l$$

with $l' = \kappa x'.Cut_0(x.t', u)$.
If F is negative, then $E = F$, $l = \epsilon$ and Λ contains only the I_l rule. We have thus

$$\cfrac{}{\Gamma \Downarrow F \vdash l' : E}\ I_l$$

with $l' = \epsilon$.
Now if $F_1 \supset \cdots \supset F_m \supset F \neq E'$, then we have

$$\cfrac{\cfrac{\overset{\Lambda_j'}{\Gamma \vdash q_j' : F_j \Downarrow} \qquad \overset{\Lambda'}{\Gamma \Downarrow F \vdash l' : E}}{\Gamma \Downarrow F_1 \supset \cdots \supset F_m \supset F \vdash q_1' :: \cdots :: q_m' :: l' : E}\ \supset L \times m}{\Gamma \vdash z \frown (q_1' :: \cdots :: q_m' :: l') : E}\ D_l$$

If $F_1 \supset \cdots \supset F_m \supset F = E'$, then $m = 0$ and $F = E'$. Since E' is negative, F is negative. Hence, Λ contains only the I_l rule, $E = F = E'$ and $l = \epsilon$. We have thus $t = z \frown \epsilon$.
If $z \neq x$, we have

$$\cfrac{\cfrac{}{\Gamma \Downarrow F \vdash \epsilon : E}\ I_l}{\Gamma \vdash z \frown \epsilon : E}\ D_l$$

If $z = x$, we have

$$\overset{\Pi_1}{\Gamma \vdash u : E}$$

Cut$_1$:

$$\cfrac{\overset{\Pi_1}{\Gamma \vdash u : B \Uparrow} \qquad \overset{\Pi_2}{\Gamma, x : B \vdash t : E}}{\Gamma \vdash Cut_1(x.t, u) : E}\ Cut_1$$

– B is atomic: we have $\Pi_1 = \cfrac{\overset{\Pi_1'}{\Gamma \vdash u : B}}{\Gamma \vdash u : B \Uparrow}\ S_r$ and thus $\cfrac{\overset{\Pi_1'}{\Gamma \vdash u : B} \qquad \overset{\Pi_2}{\Gamma, x : B \vdash t : E}}{\Gamma \vdash Cut_0(x.t, u) : E}\ Cut_0$
– B is an implication $B_1 \supset \cdots \supset B_n \supset B_{n+1}$ with $n \geq 0$ and B_{n+1} atomic:
We have

$$\Pi_1 = \cfrac{\overset{\Pi_1'}{\Gamma, x_1 : B_1, \cdots, x_n : B_n \vdash u' : B_{n+1}}}{\Gamma \vdash u : B \Uparrow}\ \supset R/S_l \times n, S_r$$

with $u = \lambda x_1 \cdots \lambda x_n.u'$.

We follow the same reasoning about Π_2 as in the case of Cut_0. The only difference lies in the case that Π_2 ends with a D_l rule on $F_1 \supset \cdots \supset F_m \supset F$ and that $F_1 \supset \cdots \supset F_m \supset F = B$. In this case, we have $m = n$, $F_i = B_i$ for $1 \leq i \leq n$ and $F = B_{n+1}$.
Now suppose that we have a proof

$$\begin{array}{c}\Xi \\ \Gamma, \Delta, x_j : B_j \vdash t^* : B_{n+1}\end{array}$$

and consider

$$\begin{array}{c}\Lambda'_j \\ \Gamma \vdash q'_j : B_j \Downarrow\end{array}$$

If B_j is positive, then Λ'_j contains only the I_r rule, q'_j is a variable and $q'_j : B_j \in \Gamma$ and thus a proof

$$\begin{array}{c}\Xi[x_j \mapsto q'_j] \\ \Gamma, \Delta \vdash t^*[x_j \mapsto q'_j] : B_{n+1}\end{array}$$

If B_j is negative, then Λ'_j ends with the R_r rule. We have thus

$$\dfrac{\Gamma, \Delta \vdash q'_j : B_j \Uparrow \qquad \begin{array}{c}\Xi \\ \Gamma, \Delta, x_j : B_j \vdash t^* : B_{n+1}\end{array}}{\Gamma, \Delta \vdash Cut_1(x_j.t^*, q'_j) : B_{n+1}}\ Cut_1$$

By considering the proof Π'_1, the proofs $\Lambda'_j(1 \leq \jmath \leq n)$ and by applying the above step, we obtain a proof

$$\begin{array}{c}\Lambda^* \\ \Gamma \vdash u'' : B_{n+1}\end{array}$$

Now consider the proof

$$\begin{array}{c}\Lambda' \\ \Gamma \Downarrow B_{n+1} \vdash l' : E\end{array}$$

If B_{n+1} is negative, then $E = B_{n+1}$, $l' = \epsilon$ and Λ' contains only the I_l rule. We have thus

$$\begin{array}{c}\Lambda^* \\ \Gamma \vdash u'' : E\end{array}$$

If B_{n+1} is positive, then we have

$$\Lambda' = \dfrac{\begin{array}{c}\Lambda'' \\ \Gamma, y : B_{n+1} \vdash t' : E\end{array}}{\Gamma \Downarrow B_{n+1} \vdash l' : E}\ R_l/S_l$$

where $l' = \kappa y.t'$. Hence, we obtain

$$\dfrac{\begin{array}{c}\Lambda^* \\ \Gamma \vdash u'' : B_{n+1}\end{array} \qquad \begin{array}{c}\Lambda'' \\ \Gamma, y : B_{n+1} \vdash t' : E\end{array}}{\Gamma \vdash Cut_0(y.t', u'') : E}\ Cut_0$$