Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis*

J. B. Wells[1] and Boris Yakobowski[2]

[1] Heriot-Watt University, http://www.macs.hw.ac.uk/~jbw/
[2] ENS Lyon, http://www.yakobowski.org/cs/

Abstract. For use in earlier approaches to automated module interface adaptation, we seek a restricted form of program synthesis. Given some typing assumptions and a desired result type, we wish to automatically build a number of program fragments of this chosen typing, using functions and values available in the given typing environment. We call this problem *term enumeration*. To solve the problem, we use the Curry-Howard correspondence (propositions-as-types, proofs-as-programs) to transform it into a *proof enumeration* problem for an intuitionistic logic calculus. We formally study proof enumeration and counting in this calculus. We prove that proof counting is solvable and give an algorithm to solve it. This in turn yields a proof enumeration algorithm.

1 Introduction

1.1 Background and motivation

Researchers have recently expressed interest [9, 10, 1] in type-directed program synthesis that outputs terms of a desired goal typing (i.e., environment of type assumptions and result type) using the values (possibly functions) available in the type environment. These terms are typically wanted for use in simple glue code that adapts one module interface to another, overcoming simple interface differences. There are usually many terms of the goal typing, with many computational behaviors, and only some will satisfy all the user's criteria. To find terms of the goal typing that satisfy all the criteria, it is desirable to systematically enumerate terms of the typing. The enumerated terms can then be filtered [9, 10], possibly with user assistance [1], to find the most suitable ones.

Higher-order typed languages (e.g., the ML family) are suitable for this kind of synthesis. They have expressive type systems that allow specifying precise goals. They also support easily composing and decomposing functions, tuples, and tagged variants, which can accomplish most of what is needed for the kind of simple interface adaptation we envision.

* Supported by grants: EC FP5/IST/FET IST-2001-33477 "DART", EPSRC GR/L 41545/01, NSF 0113193 (ITR), Sun Microsystems EDUD-7826-990410-US.

1.2 Applications

Both the AxML module adaptation approach [9, 10] and work on signature subtyping modulo isomorphisms [1] do whole module adaptation through the use of higher-order ML functors.

In AxML, term enumeration is mainly needed to fill in unspecified holes in adaptation code and the main adaptation work is done by other mechanisms. Term enumeration is useful because an unspecified hole may indicate that the programmer has not thought things through and they might benefit from seeing possible alternatives for filling the hole. This will mainly be useful when the alternatives are small and have somewhat distinct behavior, so a systematic breadth-first enumeration is expected to be best and enumerating many large chunks of code would likely be less useful.

In the work on signature subtyping modulo isomorphisms, requirements for the calculus are quite light: only arrow types (and a subtyping rule) are needed. Typical examples involve applying a functor to a pre-existing module, in order to get a module having the same signature as the result of the functor. For example, we might compose a functor resulting in a map over a given type with a module containing a generic comparison function.

1.3 Possible approaches to term enumeration

Program synthesis such as term enumeration seeks to find functions with some desired behavior, which is similar to library *retrieval*. Closer to our task, some retrieval systems also *compose* functions available in the library (see [10] for discussion), but are not suitable for enumeration. Research on *type inhabitation* [3, 13, 17] is related, but is mostly concerned with the *theoretical* issue of the number of terms in a typing (mainly whether there is at least 1), and the resulting enumeration algorithms are overly inefficient.

The most closely related work is on *proof search*. Although most of this work focuses on yes/no answers to theorem proving queries or on building individual proofs, there has been some work on proof enumeration in various logics [6, 14, 16]. With constructive logics, we can use the Curry-Howard correspondence to generate terms from the proofs of a formula. We follow this approach here.

1.4 Overview

We explain in sec. 2 that the existing calculus LJT is the most suited to our task and we modify it slightly in sec. 4 to make the even more suitable $\mathsf{LJT}^{\mathsf{Enum}}$. Next, we present in sec. 5 a graph representation of proofs and use it to show solvability of proof counting. In sec. 6, we present Count, a direct proof counting algorithm, and outline proof enumeration. We then discuss in sec. 7 the links between proof counting and term enumeration and add proof terms to $\mathsf{LJT}^{\mathsf{Enum}}$.

1.5 Acknowledgements

We are grateful to Christian Haack, Daniel Hirschkoff, and the anonymous referees for their helpful comments on earlier versions.

2 Which calculus for proof enumeration?

As already mentioned, proof enumeration is defined as the enumeration of all the proofs of a formula, as opposed to finding only one proof. Using the Curry-Howard correspondence, term enumeration can be reduced to proof enumeration; but for that approach to be usable, there must exist some guaranties on the correspondence. For example, 1-∞ correspondences are unsuitable, because we might have to examine an infinity of proofs to find different program fragments.

In our case, it is important to find a calculus in which the proofs are in bijection with normal λ-terms, or equivalently with the set of normal terms in natural deduction style. Dyckhoff and Pinto [6] provide a survey of various calculi usable for proof enumeration. They argue that "*the appropriate proof-search calculi are those that have not only the syntax-directed features of Gentzen-style sequent calculi but also a natural 1–1 correspondence between the derivations and the real objects of interest, normal natural deductions*" and we agree with their analysis. Unfortunately, calculi having these properties are quite rare.

For example, a sequent calculus such as Gentzen's LJ does not meet the previous criteria. Indeed, due to possible permutations in the proofs, or to the use of cut rules, two proofs can be associated to the same term. In fact, it has been long known that 2 proofs in LJ are "the same", meaning that they are equivalent to the same normal deduction proof in NJ, if they are interpermutable. As a result, we have to consider cut-free and *permutation-free* calculi.

Historically, the first calculus having those properties is Herbelin's LJT [11]. Proofs in LJT are in bijection with the terms of the simply typed λ-calculus. Afterwards, Herbelin introduced LKT [12], which is based on Gentzen's classical calculus LK, and Pinto and Dyckhoff [16] proposed two other calculi for systems with dependent types. Of all these calculi, LJT is better adapted to our purpose, because the additional features in the three others do not help our task.

The permutation-free property of LJT is achieved by adding in each sequent a special place, called a *stoup*, used to focus the proof. The stoup can either be empty or filled by one variable. Once the stoup is full, deductions can only be made based on its content, and it cannot be emptied easily. The content of the stoup is interpreted as the head variable in the standard λ-calculus.

All the sequents provable in LJ are provable in LJT. The cut-free version of LJT is a sequent calculus which enjoys the subformula property, and is syntax-directed, with few sources of non determinism; LJT also enjoys a cut elimination theorem [11, 7], so we can restrict ourselves to considering only cut-free proofs. Finally, as was needed, while the traditional proof terms in LJ correspond to the simply-typed λ-terms, the proof terms in the cut-free version of LJT are in bijection with the simply-typed λ-terms in normal form, so all interesting terms may potentially be found.

3 Mathematical preliminaries

Given a set E, let $\mathsf{Set}(E)$ be the set of all subsets of E. Let a *multiset* over E be a function from E to $\mathbb{N}$ (the natural numbers); if $\mathcal{M}$ is a multiset, we say

that $m \in \mathcal{M}$ iff $\mathcal{M}(m) > 0$. A multiset $\mathcal{M}$ is *finite* iff $\{ m \mid m \in \mathcal{M} \}$ is finite. Let $\mathsf{MSet}(E)$ be the set of all multisets over E. Let $\mathsf{FinMSet}(E)$ be the set of all finite multisets over E. We write multiset literals using the same notation as sets.

Multiset union is defined as usual by $(\mathcal{M}_1 \uplus \mathcal{M}_2)(x) = \mathcal{M}_1(x) + \mathcal{M}_2(x)$. A "set-like" multiset union is defined by $(\mathcal{M}_1 \cup\!\!\cup \mathcal{M}_2)(x) = \max(\mathcal{M}_1(x), \mathcal{M}_2(x))$. Let $\mathcal{S}$ range over the names Set and MSet. Let $\cup_{\mathsf{Set}} = \cup\!\!\cup$ and $\cup_{\mathsf{MSet}} = \uplus$.

We extend the arithmetic operators $+$ and $\times$ and the relation $\leq$ to $\mathbb{N} \cup \{\infty\}$ using the usual arithmetic rules for members of $\mathbb{N}$, and by letting $n + \infty = \infty$, $n \times \infty = \infty$ if $n \neq 0$, $0 \times \infty = 0$, and $n \leq \infty$. Also, as usual let $\Sigma_{x \in \emptyset} v(x) = 0$ and let $\Pi_{x \in \emptyset} v(x) = 1$.

Given a set S, a directed graph G over S is a pair (V, E) where $V \subset S$ and $E \subset S \times S$. The elements of V are the vertexes of G, and those of E are the edges of G. Given a graph $G = (V, E)$, let $\mathsf{succ}(G, v) = \{v' \in V \mid (v, v') \in E\}$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, let $G_1 \cup G_2$ be $(V_1 \cup V_2, E_1 \cup E_2)$.

We represent mathematical functions as sets of pairs. Let the *domain* of a function f be $\mathsf{Dom}(f) = \{ x \mid (x, y) \in f \}$. To modify functions, we write $f, x : v$ for $(f \setminus \{ (x, y) \mid (x, y) \in f \}) \cup \{(x, v)\}$.

4 The calculus $\mathsf{LJT}^{\mathsf{Enum}}$

In this section, we present $\mathsf{LJT}^{\mathsf{Enum}}$, a slightly modified version of LJT more suitable for term enumeration. The following pseudo-grammars define the syntax.

$$
\begin{array}{rl}
Q \in & \mathsf{Propositional\text{-}Variables} ::= Q_i \\
X, Y \in & \mathsf{Basic\text{-}Propositions} ::= Q \mid Q[A_1, \ldots, A_n] \\
A, B \in & \mathsf{Formulas} ::= X \mid A_1 \rightarrow A_2 \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \\
A^? \in & \mathsf{Stoups} ::= A \mid \bullet \\
\Gamma \in & \mathsf{Environments}_{\mathsf{MSet}} = \mathsf{FinMSet}(\mathsf{Formulas}) \\
s \in & \mathsf{Sequents}_{\mathsf{MSet}} ::= \Gamma; A^? \vdash B
\end{array}
$$

Let also $\mathsf{Environments}_{\mathsf{Set}} = \{\Gamma \in \mathsf{Environments}_{\mathsf{MSet}} \mid \forall A \in \mathsf{Formulas}, \Gamma(A) \leq 1\}$. Let $\mathsf{Sequents}_{\mathsf{Set}}$ be the subset of $\mathsf{Sequents}_{\mathsf{MSet}}$ such that the environment of each sequent is in $\mathsf{Environments}_{\mathsf{Set}}$. The symbol $\bullet$ is the empty stoup.

Basic propositions which are not propositional variables are used to encode parameterized ML types, such as `list`. For example, if `int` is encoded as A and `list` as B, `int list` is encoded as $B[A]$. Note that we do not yet support polymorphism as in $\forall\alpha.\ \alpha$ `list`. Separate functions for handling `int list` or `bool list` must be supplied in the environment.

We present the rules of $\mathsf{LJT}^{\mathsf{Enum}}_{\mathcal{S}}$ in Fig. 1, which are basically the cut-free rules of LJT. (The appendix explains the few differences between LJT and $\mathsf{LJT}^{\mathsf{Enum}}_{\mathcal{S}}$.)

The rules which add elements in the environment are parameterized by the operation to use. The two systems $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Set}}$ and $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{MSet}}$ prove essentially the same judgements, but with possibly different proof trees. This distinction helps in analyzing the problem of term enumeration and devising our solution. These points will be developed in sections 5 and 6.

Axiom rule

$$\frac{}{\Gamma; X \vdash X}\ \textsc{Ax}$$

Contraction rule

$$\frac{\Gamma \uplus \{A\}; A \vdash B}{\Gamma \uplus \{A\}; \bullet \vdash B}\ \textsc{Cont}(A)$$

Left implication rule

$$\frac{\Gamma; \bullet \vdash A \qquad \Gamma; B \vdash C}{\Gamma; A \to B \vdash C}\ \textsc{Imp}_{\mathrm{L}}$$

Right implication rule

$$\frac{\Gamma \cup_{\mathsf{S}} \{A\}; \bullet \vdash B}{\Gamma; \bullet \vdash A \to B}\ \textsc{Imp}_{\mathrm{R}}$$

Left conjunction rule

$$\frac{\Gamma; A_i \vdash B}{\Gamma; A_1 \wedge A_2 \vdash B}\ \textsc{And}_{\mathrm{L}_i}$$

Right conjunction rule

$$\frac{\Gamma; \bullet \vdash A \qquad \Gamma; \bullet \vdash B}{\Gamma; \bullet \vdash A \wedge B}\ \textsc{And}_{\mathrm{R}}$$

Left disjunction rule

$$\frac{\Gamma \cup_{\mathsf{S}} \{A\}; \bullet \vdash C \qquad \Gamma \cup_{\mathsf{S}} \{B\}; \bullet \vdash C}{\Gamma; A \vee B \vdash C}\ \textsc{Or}_{\mathrm{L}}$$

Right disjunction rule

$$\frac{\Gamma; \bullet \vdash A_i}{\Gamma; \bullet \vdash A_1 \vee A_2}\ \textsc{Or}_{\mathrm{R}_i}$$

Fig. 1. Rules of $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{S}}$ $(i \in \{1, 2\})$

5 The proof counting problem

In this section, we formally study the problems of proof counting in $\mathsf{LJT}^{\mathsf{Enum}}$. We interpret sequent resolution as a graph problem. From that we prove that finding the number of proofs (which is ∞ if there are an infinite number of proofs) of a sequent is computable.

Let $\mathsf{C}_{\mathsf{S}}(s)$ be the number of proofs of a sequent s in $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{S}}$. $\mathsf{C}_{\mathsf{MSet}}(s)$ is strongly related to the number of different terms which can be obtained from the proofs of s; section 7.2 will discuss this. Although apparently less interesting, $\mathsf{C}_{\mathsf{Set}}(s)$ is much easier to compute, and can help in finding $\mathsf{C}_{\mathsf{MSet}}(s)$.

5.1 A graph representation of possible proofs

We start by defining the notion of applicable rule to a sequent. Let R be the set of rules $\mathsf{R} = \{\textsc{Ax}, \textsc{Imp}_{\mathrm{L}}, \textsc{Imp}_{\mathrm{R}}, \textsc{And}_{\mathrm{L}_1}, \textsc{And}_{\mathrm{L}_2}, \textsc{And}_{\mathrm{R}}, \textsc{Or}_{\mathrm{L}}, \textsc{Or}_{\mathrm{R}_1}, \textsc{Or}_{\mathrm{R}_2}, \textsc{Cont}(A) \mid A \in \mathsf{Formulas}\}$. Let r range over R.

A rule r with conclusion c is *applicable* to a sequent s iff, viewing the basic propositions and formulas in r as meta-variables, there is a substitution σ from these meta-variables to basic propositions or formulas such that $\sigma(c) = s$. If the rule is $\textsc{Cont}(A)$, the formula A must be the one chosen from the environment Γ. Let $\mathsf{RA}(s)$ be the set of rules applicable to a sequent s. Let the valid sequent/rule pairs be $\mathsf{VP}_{\mathsf{S}} = \{(s, r) \mid s \in \mathsf{Sequents}_{\mathsf{S}}, r \in \mathsf{RA}(s)\}$. Let τ range over VP_{S}.

Given a sequent s and a rule r applicable to s via a substitution σ, let $\mathsf{Pr}_{\mathsf{S}}(s, r, i)$ be the ith premise of $\sigma(r)$ using $\cup_{\mathsf{S}}$ as the combining operator on the environment if r has at least i premises.

Definition 5.1. *Let* $\mathsf{G}_{\mathsf{S}} = (\mathsf{V}_{\mathsf{S}}, \mathsf{E}_{\mathsf{S}})$ *be the directed graph of all possible sequents and rule uses in* $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{S}}$ *defined by:*

- $\mathsf{V}_{\mathcal{S}} = \mathsf{Sequents}_{\mathcal{S}} \cup \mathsf{VP}_{\mathcal{S}}$.
- $\mathsf{E}_{\mathcal{S}_1} = \{(s,\, (s,\, r)) \mid s \in \mathsf{Sequents}_{\mathcal{S}},\ r \in \mathsf{RA}(s)\}$.
- $\mathsf{E}_{\mathcal{S}_2} = \{((s,r), s', n) \mid n \in \mathbb{N},\ s' = \mathsf{Pr}_{\mathcal{S}}(s,r,n)\}$.
- $\mathsf{E}_{\mathcal{S}} = \mathsf{E}_{\mathcal{S}_1} \cup \mathsf{E}_{\mathcal{S}_2}$.

The elements of $\mathsf{V}_{\mathcal{S}}$ which are in $\mathsf{Sequents}_{\mathcal{S}}$ are called sequent vertexes. Their outgoing edges (which are in $\mathsf{E}_{\mathcal{S}_1}$) go to valid pairs. The elements of $\mathsf{V}_{\mathcal{S}}$ which are in $\mathsf{VP}_{\mathcal{S}}$ are called rule-use vertexes. Their outgoing edges (which are in $\mathsf{E}_{\mathcal{S}_2}$) go to the sequents which are the premises of the rule use. On each outgoing edge we add a number indicating which premise we are considering (needed only when there is more than one premise). An example is provided in Fig. 2.

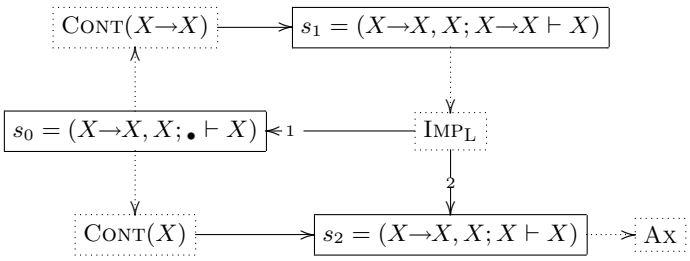

Fig. 2.
$\mathsf{G}_{\mathsf{Set}}(s_0) = \mathsf{G}_{\mathsf{MSet}}(s_0)$

The *lowering* of a multiset $\mathcal{M}$ to a "set-like" multiset $\lfloor\mathcal{M}\rfloor$ is defined such that $\lfloor\mathcal{M}\rfloor(x) = \min(1, \mathcal{M}(x))$. Let $\lfloor(\Gamma; A^? \vdash B)\rfloor = (\lfloor\Gamma\rfloor; A^? \vdash B)$, let $\lfloor(s,r)\rfloor = (\lfloor s\rfloor, r)$, let $\lfloor(s,\tau)\rfloor = (\lfloor s\rfloor, \lfloor\tau\rfloor)$, and let $\lfloor(\tau,s,i)\rfloor = (\lfloor\tau\rfloor, \lfloor s\rfloor, i)$. Given any set W, let $\lfloor(W)\rfloor = \{\lfloor w\rfloor \mid w \in W\}$. For graphs, let $\lfloor(V,E)\rfloor = (\lfloor V\rfloor, \lfloor E\rfloor)$. Note that $\lfloor\mathsf{G}_{\mathsf{MSet}}\rfloor = \mathsf{G}_{\mathsf{Set}}$.

A graph $g = (V,E)$ is an $\mathcal{S}$-*subgraph* iff $V \subseteq \mathsf{V}_{\mathcal{S}}$, $E \subseteq \mathsf{E}_{\mathcal{S}}$, and $s, \tau \in V$ whenever $(s,\tau) \in E$ or $(\tau, s, i) \in E$. An $\mathcal{S}$-subgraph $g = (V,E)$ is *valid* iff for every $\tau = (s,r) \in V$ where r has n premises, $(\tau, \mathsf{Pr}_{\mathcal{S}}(s,r,i), i) \in E$ for $1 \le i \le n$.

In addition to lowering, we define *raising* of entities of $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Set}}$ to $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{MSet}}$. A subgraph $g = (V,S)$ of $\mathsf{G}_{\mathsf{Set}}$ can be used as a template to build a corresponding subgraph g' of $\mathsf{G}_{\mathsf{MSet}}$, starting from an MSet-sequent s such that $\lfloor s\rfloor \in V$. We recursively follow the edges of g from $\lfloor s\rfloor$ and build corresponding edges in g' from s, using $\uplus$ instead of $\cup$ as the environment combining operator. The resulting subgraph g' may be infinite, even if g is finite.

Formally, raising is defined as follows. Given a valid Set-subgraph $g = (V,E)$ and an MSet-sequent s such that $\lfloor s\rfloor \in V$, let $\mathsf{raise}(s,g)$ be the smallest valid MSet-subgraph $g' = (V', E')$ such that (1) $s \in V'$ and (2) for every $s' \in V'$ if $(\lfloor s'\rfloor, r) \in V$ then $(s', r) \in V'$. Note that parts of g that are not reachable from $\lfloor s\rfloor$ do not have an image in $\mathsf{raise}(s,g)$.

Given a sequent s, let $\mathsf{G}_{\mathcal{S}}(s)$ be the subgraph of $\mathsf{G}_{\mathcal{S}}$ containing all the sequent and rule-use vertexes reachable from s. From a practical viewpoint, $\mathsf{G}_{\mathcal{S}}(s)$ is the largest subgraph of $\mathsf{G}_{\mathcal{S}}$ that a procedure attempting to find proofs of s should have to consider. It is worth noting that in the general case, $\mathsf{G}_{\mathsf{Set}}(s)$ and

$\mathsf{G}_{\mathsf{MSet}}(s)$ may be cyclic graphs (e.g., in Fig. 2). Note that $\lfloor \mathsf{G}_{\mathsf{MSet}}(s) \rfloor = \mathsf{G}_{\mathsf{Set}}(\iota s \imath)$ and $\mathsf{raise}(s, \mathsf{G}_{\mathsf{Set}}(\iota s \imath)) = \mathsf{G}_{\mathsf{MSet}}(s)$.

Lemma 5.2 (Finiteness). $\mathsf{G}_{\mathsf{Set}}(s)$ *is always finite.* $\mathsf{G}_{\mathsf{MSet}}(s)$ *can be infinite.*

Proof. When environments are sets, it is a direct consequence of the fact that $\mathsf{LJT}^{\mathsf{Enum}}$ enjoys the subformula property. When environments are multisets, a sufficient condition for the graph to be infinite is to have in the context a function taking as an argument a function, or a disjunction. We can then find a derivation branch in which a formula can be added an arbitrary number of times in the environment, making the graph infinite. See for example Fig. 3. □

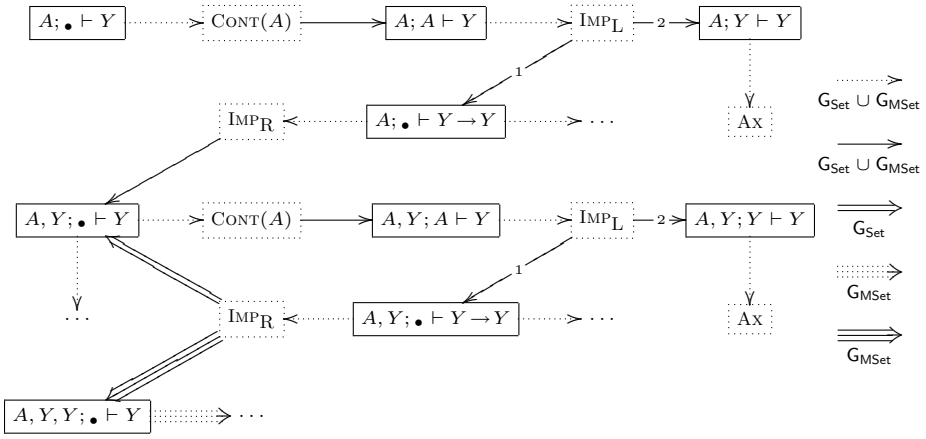

Fig. 3. A subgraph of $\mathsf{G}_{\mathsf{MSet}}(A; \bullet \vdash Y)$ and $\mathsf{G}_{\mathsf{Set}}(A; \bullet \vdash Y)$ with $A = (Y \to Y) \to Y$

5.2 Proof trees and their relationship with the graph

We now define a structure which captures exactly one proof of a sequent.

Definition 5.3 (Proof trees). *Let proof trees be given by this pseudo-grammar:*

$$T \in \mathsf{ProofTree} ::= \tau(T_1, \ldots, T_n)$$

Let $\mathsf{Seq}(s, r) = s$ *and let* $\mathsf{Seq}(\tau(T_1, \ldots, T_n)) = \mathsf{Seq}(\tau)$. *A particular proof tree* $T = (s, r)(T_1, \ldots, T_n)$ *is an* $\mathcal{S}$-proof tree *iff (1)* $s \in \mathsf{Sequents}_{\mathcal{S}}$, *(2)* $r \in \mathsf{RA}(s)$, *and (3)* r *has* n *premises and for* $1 \leq i \leq n$ *it holds that* T_i *is an* $\mathcal{S}$*-proof tree such that* $\mathsf{Seq}(T_i) = \mathsf{Pr}_{\mathcal{S}}(s, r, i)$. *We henceforth consider only* $\mathcal{S}$*-proof trees.*

We recursively fold an $\mathcal{S}$-proof tree into a $\mathcal{S}$-valid subgraph of $\mathsf{G}_{\mathcal{S}}(s)$ by:

$$\begin{array}{l}\mathsf{Fold}_{\mathcal{S}}((s,r)(T_1,\ldots,T_n)) \\ \quad = \quad (\{s,(s,r)\} \cup \{\, \mathsf{Seq}(T_i) \,|\, 1 \le i \le n \,\}, \\ \qquad\quad \{(s,(s,r))\} \cup \{\, ((s,r),\mathsf{Seq}(T_i),i) \,|\, 1 \le i \le n \,\}) \\ \qquad \cup (\bigcup_{1\le i\le n} \mathsf{Fold}_{\mathcal{S}}(T_i))\end{array}$$

To allow lowering MSet-proof trees to Set-proof trees, let $\llcorner \tau(T_1,\ldots,T_n) \lrcorner = \llcorner \tau \lrcorner(\llcorner T_1 \lrcorner,\ldots,\llcorner T_n \lrcorner)$. Similarly, a Set-proof tree T can be raised to an MSet-proof tree T' such that $\llcorner \mathsf{Seq}(T') \lrcorner = \mathsf{Seq}(T)$:

$$\begin{array}{l}\mathsf{raise}(s,(\llcorner s \lrcorner,r)(T_1,\ldots,T_n)) \\ \quad = (s,r)(\mathsf{raise}(\mathsf{Pr}_{\mathsf{MSet}}(s,r,1),T_1),\ldots,\mathsf{raise}(\mathsf{Pr}_{\mathsf{MSet}}(s,r,n),T_n))\end{array}$$

The $\mathsf{Fold}_{\mathcal{S}}$ operation commutes with lowering and raising. If T is a Set-proof tree such that $\mathsf{Seq}(T) = \llcorner s \lrcorner$, then $\mathsf{Fold}_{\mathsf{MSet}}(\mathsf{raise}(s,T)) = \mathsf{raise}(s,\mathsf{Fold}_{\mathsf{Set}}(T))$. If T is an MSet-proof tree, then $\mathsf{Fold}_{\mathsf{Set}}(\llcorner T \lrcorner) = \llcorner \mathsf{Fold}_{\mathsf{MSet}}(T) \lrcorner$.

An $\mathcal{S}$-proof tree T is *acyclic* iff $\mathsf{Fold}_{\mathcal{S}}(T)$ is acyclic. Given an acyclic $\mathcal{S}$-proof tree T, there are only a finite number (possibly more than 1) of $\mathcal{S}$-proof trees T' such that $\mathsf{Fold}_{\mathcal{S}}(T') = \mathsf{Fold}_{\mathcal{S}}(T)$. Given a sequent s for which $\mathsf{G}_{\mathcal{S}}(s)$ is finite, it is possible to count the number of acyclic $\mathcal{S}$-proof trees for s, by a simple brute force enumeration (there are only a finite possible number of them).

An $\mathcal{S}$-proof tree T is *cyclic* iff $\mathsf{Fold}_{\mathcal{S}}(T)$ is cyclic. Given a cyclic $\mathcal{S}$-proof tree T, there are an infinite number of $\mathcal{S}$-proof trees T' such that $\mathsf{Fold}_{\mathcal{S}}(T') = \mathsf{Fold}_{\mathcal{S}}(T)$. This follows from the fact that in a cyclic $\mathcal{S}$-proof tree, the proof of some sequent s depends on a smaller proof of s. Thus, each time we find a proof of s, we can build a new, bigger (with respect to the height of the proof tree) proof of s, by unfolding the proof already found.

The raising of an acyclic Set-proof tree is an acyclic MSet-proof tree, and the lowering of a cyclic MSet-proof tree is a cyclic Set-proof tree. But the lowering of an acyclic MSet-proof tree can be a cyclic Set-proof tree. Similarly, the raisng of a cyclic Set-proof tree can be an acyclic MSet-proof tree. Fig. 3 shows part of an example of the last two points.

5.3 Proof counting

Lemma 5.4. *Let s be a sequent.*

- *There are the same number of $\mathsf{LJT}^{\mathsf{Enum}}_{\mathcal{S}}$ proofs of s and $\mathcal{S}$-proof trees for s.*
- *Suppose $\mathsf{G}_{\mathcal{S}}(s)$ is finite. If there is no cyclic $\mathcal{S}$-proof tree for s, then the number of $\mathcal{S}$-proof trees for s is finite; otherwise it is infinite.*
- *Suppose $\mathsf{G}_{\mathcal{S}}(s)$ is infinite. If there is no cyclic $\mathcal{S}$-proof tree for s, then the number of $\mathcal{S}$-proofs of s can be either finite or infinite; otherwise it is infinite.*

Lemma 5.5. *Given an* MSet*-sequent s, if there exists an infinity of acyclic* MSet*-proof trees for s, then there exists a cyclic* Set*-proof tree for $\llcorner s \lrcorner$.*

Proof. We say that a proof tree is of height n iff its longest path goes through n sequent nodes. Let N be the number of Set-sequent nodes in $\mathsf{G}_{\mathsf{Set}}(\llcorner s \lrcorner)$.

We first prove that there exists an MSet-proof tree T for s of height greater than N. For this, construct a (possibly infinite in branching and number of nodes) tree BT ("big tree") by unfolding the graph $\mathsf{G}_{\mathsf{MSet}}(s)$ starting from s into a tree, choosing some arbitrary order for the rule-use children of a sequent node, and making all sequent nodes at depth N (not counting rule-use nodes and with the root sequent node at depth 1) into leaves and adding no further children beyond depth N. By construction, all MSet-proof trees of s of height less than N can be seen to be "embedded" in BT.

Now we observe that BT is finitely branching. For every sequent s' occurring in BT, there are a finite number of rule uses that can use other sequents to prove s'. This is so because R is finite except for rules of the form $\textsc{Cont}(A)$, and at most a finite number of those can apply to s' because the environment Γ of s' can mention only a finite number of distinct formulas.

Now, by König's lemma, BT contains a finite number of nodes. As a consequence, there are only a finite number of distinct MSet-proof trees embedded in BT. Thus T exists and has height $m > N$.

The Set-proof tree $\llcorner T \lrcorner$ has the same height as T, so $\llcorner T \lrcorner$ has at least one path of length m. Along this path, some Set-sequent nodes must be repeated in $\mathsf{Fold}_{\mathsf{Set}}(\llcorner T \lrcorner)$, and thus $\llcorner T \lrcorner$ is a cyclic Set-proof tree for $\llcorner s \lrcorner$. □

Theorem 5.6. *Let $s \in \mathsf{Sequents}_{\mathsf{MSet}}$. Then all of the following statements hold:*

- $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) \leq \mathsf{C}_{\mathsf{MSet}}(s)$.
- $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = \infty \iff \mathsf{C}_{\mathsf{MSet}}(s) = \infty$.
- $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = 0 \iff \mathsf{C}_{\mathsf{MSet}}(s) = 0$.

Proof. The first point is easy: for each Set-proof tree T for $\llcorner s \lrcorner$, $\mathsf{raise}(s, T)$ is a MSet-proof tree for s, and raise is injective. This also proves that $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = \infty \Rightarrow \mathsf{C}_{\mathsf{MSet}}(s) = \infty$ and $\mathsf{C}_{\mathsf{MSet}}(s) = 0 \Rightarrow \mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = 0$.

Next, suppose that $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = 0$. If there was an MSet-proof tree T for s, then $\llcorner T \lrcorner$ would be a Set-proof tree for $\llcorner s \lrcorner$ and we would have $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) \neq 0$. Absurd.

Finally suppose that $\mathsf{C}_{\mathsf{MSet}}(s) = \infty$. There are two cases: (1) There is a cyclic MSet-proof tree T for s. Then $\llcorner T \lrcorner$ is a cyclic Set-proof tree for $\llcorner s \lrcorner$; (2) There are no cyclic MSet-proof trees for s. By lemma 5.4, it means there are an infinite number of acyclic MSet-proof trees for s. Then by Lemma 5.5, there is a cyclic Set-proof tree for $\llcorner s \lrcorner$. In both cases, by Lemma 5.4, $\mathsf{C}_{\mathsf{Set}}(\llcorner s \lrcorner) = \infty$. □

Theorem 5.7. *Proof counting is computable for $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Set}}$ and $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{MSet}}$.*

Proof. The following algorithm CountNaive counts the proofs of a sequent s:

1. Build $\mathsf{G}_{\mathsf{Set}}(s)$; by Lemma 5.2, it is finite.
2. Search for a cyclic Set-proof tree for s. For this, use the same exhaustive enumeration as when searching for acyclic ones, but stop as soon as a cyclic one is found. If a cyclic Set-proof tree is found, then return $\infty = \mathsf{C}_{\mathsf{MSet}}(s) = \mathsf{C}_{\mathsf{Set}}(s)$ (by Theorem 5.6).

3. Otherwise $\mathsf{C}_{\mathsf{MSet}}(s)$ and $\mathsf{C}_{\mathsf{Set}}(s)$ are finite, by Theorem 5.6. If we are searching for $\mathsf{C}_{\mathsf{Set}}(s)$, return the number of Set-proof trees for s found by the exhaustive enumeration in the previous step.
4. Otherwise, we are searching for $\mathsf{C}_{\mathsf{MSet}}(s)$. Build a restricted (and finite) subgraph g of $\mathsf{G}_{\mathsf{MSet}}(s)$ containing all the foldings of the MSet-proof trees for s. For this, start at s and do a breadth-first exploration. At each new node s' visited, check whether or not it is provable, by finding the number of proofs of $\llcorner s' \lrcorner$ in $\mathsf{G}_{\mathsf{Set}}(s)$, which is the number of Set-proof trees for $\llcorner s' \lrcorner$ (indeed, $\mathsf{G}_{\mathsf{Set}}(\llcorner s' \lrcorner) \subseteq \mathsf{G}_{\mathsf{Set}}(s)$ and thus cannot contain a cyclic Set-proof tree). If $\llcorner s' \lrcorner$ is unprovable, so is s' by Theorem 5.6; do not explore its successors. Because there are no arbitrarily large acyclic MSet-proof trees for s (by Lemma 5.5), g is finite and this process terminates.
5. Find the number of MSet-proof trees for s whose foldings are in g by exhaustive enumeration. By construction, it is $\mathsf{C}_{\mathsf{MSet}}(s)$. □

5.4 The generality of the idea

Our approach (using $\mathsf{G}_{\mathsf{Set}}$ to study $\mathsf{G}_{\mathsf{MSet}}$) resembles a static analysis where instead of considering the number of times a formula is present in the environment, we consider only its presence or absence. That property is interesting because provability does not depend on duplicate formulas in the environment. In our case, proof counting is also compatible with our simplifying hypothesis (because $\mathsf{C}_{\mathsf{Set}}(s) = \infty \Rightarrow \mathsf{C}_{\mathsf{MSet}}(s) = \infty$). This idea is quite general because it is usable in every calculus in which the environment only increases.

6 An algorithm for counting and enumerating proofs in $\mathsf{LJT}^{\mathsf{Enum}}$

The algorithm CountNaive could theoretically be used to find the number of proofs of a sequent. Unfortunately, it is overly inefficient. In this section we propose Count, a more efficient algorithm to compute $\mathsf{C}_{\mathbb{S}}(s)$. We also link proof counting to proof enumeration.

6.1 Underlying ideas

The main inefficiency of CountNaive is that it does not exploit the inductive structure of proof trees. Indeed, the number of proofs of a sequent vertex is the sum of the number of proofs of its successors, and the number of proofs of a rule-use vertex is the product of the number of proofs of its successors. That simple definition cannot be trivially computed, because a proof for a sequent s can use inside itself another proof of s; instead we must explicitly check for loops. As a consequence, instead of returning $\mathsf{C}_{\mathbb{S}}(s)$, we return equations verified by $\mathsf{C}_{\mathbb{S}}(s')$, for all the s' in $\mathsf{G}_{\mathbb{S}}(s)$.

Consider for example Fig. 2. The equations verified by $\mathsf{C}_{\mathcal{S}}(s_0)$, $\mathsf{C}_{\mathcal{S}}(s_1)$ and $\mathsf{C}_{\mathcal{S}}(s_2)$ are:

$$\begin{aligned}\mathsf{C}_{\mathcal{S}}(s_0) &= \mathsf{C}_{\mathcal{S}}(s_1) + \mathsf{C}_{\mathcal{S}}(s_2)\\ \mathsf{C}_{\mathcal{S}}(s_1) &= \mathsf{C}_{\mathcal{S}}(s_0) \cdot \mathsf{C}_{\mathcal{S}}(s_2)\\ \mathsf{C}_{\mathcal{S}}(s_2) &= 1\end{aligned}$$

Afterward, this set of equations must be solved, using standard mathematical reasoning. But we are only interested in the smallest solutions. Indeed, consider the system $\mathsf{C}_{\mathcal{S}}(s) = \mathsf{C}_{\mathcal{S}}(s'), \mathsf{C}_{\mathcal{S}}(s') = \mathsf{C}_{\mathcal{S}}(s)$. All the solutions $\mathsf{C}_{\mathcal{S}}(s) = \mathsf{C}_{\mathcal{S}}(s') = k$ are mathematically acceptable, but only the solution $\mathsf{C}_{\mathcal{S}}(s) = \mathsf{C}_{\mathcal{S}}(s') = 0$ counts the valid finite proof trees (none in this case).

Formally, these are polynomial equations over $\mathbb{N} \cup \{\infty\}$. An algorithm for finding the smallest solution of such systems of polynomial equations has already been given by Zaionc [17].

6.2 Formal description of the algorithm Count

An exploration of a sequent s is complete when all the subgraphs of $\mathsf{G}_{\mathcal{S}}(s)$ which could possibly lead to finding a proof have been considered. A complete exploration of $\mathsf{G}_{\mathsf{MSet}}(s)$ is not always possible, because it can be infinite. For this reason, we suppose the existence of a procedure Oracle which in the case of $\mathcal{S} = \mathsf{MSet}$ can calculate and return the value of $\mathsf{C}_{\mathsf{Set}}(s)$ (justified by theorem 5.6), although if $\mathsf{C}_{\mathsf{Set}}(s) = \infty$ we may deliberately continue exploring $\mathsf{G}_{\mathsf{MSet}}(s)$ when enumerating proofs instead of just counting. We can also use the oracle to deliberately cut off the search early when we have enumerated enough proofs.

We also suppose the existence of an algorithm Solve which takes as input a system of polynomials over $\mathbb{N} \cup \{\infty\}$, and returns as result the least solution of the system; the result should be a function from the variables used in the polynomials to their values in the solution.

In order to find $\mathsf{C}_{\mathcal{S}}(s)$, the algorithm CountSequent presented below first gathers polynomial equations verified by the sequents present in $\mathsf{G}_{\mathcal{S}}(s)$ and then uses Solve to solve the resulting system. In the polynomials, for each sequent $s' \in \mathsf{G}_{\mathcal{S}}(s)$ we use the variable $\mathsf{c}_{s'}$ to stand for $\mathsf{C}_{\mathcal{S}}(s')$.

CountSequent$(\mathcal{S}, R, s)$
1 **if** $s \in \mathsf{Dom}(R)$ **then return** R
2 **match** Oracle$(\mathcal{S}, s)$ **with**
3 $\quad \mid 0 \Rightarrow$ **return** $\{(s, 0)\} \cup R$
4 $\quad \mid \infty \Rightarrow$ **return** $\{(s, \infty)\} \cup R$
5 $v \leftarrow \sum_{\tau \in \mathsf{succ}(\mathsf{G}_{\mathcal{S}}, s)} \prod_{s' \in \mathsf{succ}(\mathsf{G}_{\mathcal{S}}, \tau)} \mathsf{c}_{s'}$
6 $R' \leftarrow \{(\mathsf{c}_s, v)\} \cup R$
7 $L \leftarrow \{\, s' \mid s' \in \mathsf{succ}(\mathsf{G}_{\mathcal{S}}, \tau), \tau \in \mathsf{succ}(\mathsf{G}_{\mathcal{S}}, s) \,\}$
8 **return** CountSet$(\mathcal{S}, R', L)$

```
CountSet(𝔖, R, L)
 1  match L with
 2    | ∅ ⇒ return R
 3    | {s} ∪ L' ⇒
 4      R' ← CountSequent(𝔖, R, s)
 5      return CountSet(𝔖, R', L')

Count(𝔖, s)
 1  R ← CountSequent(𝔖, {}, s)
 2  return (Solve(R))(c_s)
```

With a correctly choosen oracle, the algorithm always terminates. Following the results from Section 5, valid oracles would be:

- The function which always answers "No answer" in the Set case; termination is guaranteed by the finiteness of $\mathsf{G}_{\mathfrak{S}}(s)$ anyway.
- Count called with $\mathfrak{S} = \mathsf{Set}$ in the MSet case. This follows from Theorem 5.6.

$\textsc{Count}(\mathfrak{S}, s)$ returns exactly $\mathsf{C}_{\mathfrak{S}}(s)$ given a valid oracle as described just above; otherwise, the value returned is a lower bound on $\mathsf{C}_{\mathfrak{S}}(s')$.

To check the feasibility of our proof counting algorithm, we have built a completely working implementation. We present in figure 5 (p. 17) its output on an example. After each sequent, the number of proofs of that sequent is indicated. Unlike the examples presented in section 5, which were hand-made, this example is automatically[3] generated.

Our implementation uses various improvements over the algorithm presented here. For example, once a count of 0 is found in calculating a product, we do not explore the other sequents whose counts are the other factors in the product. Also, instead of calling Solve on the whole set of equations, is is more efficient to call it on all the strongly connected components of the equations, which can be found while exploring the graph in CountSequent.

6.3 Links between proof counting and proof enumeration

Exhaustive proof enumeration in $\mathsf{G}_{\mathfrak{S}}$ could be done by a breadth-first traversal of $\mathsf{G}_{\mathfrak{S}}$ to find proof trees, but that is inefficient. In particular, some infinite subparts of $\mathsf{G}_{\mathfrak{S}}$ do not lead to the finding of a proof. Our approach using proof counting is more efficient. We stop exploring a branch whenever we find out that it contains 0 solutions, and we use the more efficient computation of $\mathsf{C}_{\mathsf{Set}}(s)$ to help when computing $\mathsf{C}_{\mathsf{MSet}}(s)$. Of course, if there are an infinite number of solutions, only a finite number of them will ever be enumerated.

[3] With some manual annotations added to get a better graph layout.

7 Proof terms

In this section, we assign proof terms to proofs in $\mathsf{LJT}^{\mathsf{Enum}}$. We also discuss the links between the number of different terms which can be found from the proofs of a sequent s and $\mathsf{C}_{\mathsf{MSet}}(s)$.

7.1 The assignment of proofs to $\overline{\lambda}$-expressions

Proofs of LJT are assigned to terms of a calculus called the $\overline{\lambda}$-calculus. Compared with Herbelin's [12], our presentation is much shorter because in our cut-free calculus we only need terms in normal form. We call our restricted version of the $\overline{\lambda}$-calculus the $\overline{\lambda}'$-calculus.

In the $\overline{\lambda}'$-calculus, the usual application constructor between terms is transformed into an application constructor between a variable and a list of arguments. So there are two sorts of $\overline{\lambda}'$-expressions: $\overline{\lambda}'$-terms and lists of arguments, defined by the following pseudo-grammars where $i \in \{1,2\}$ and $j \in \mathbb{N}$:

$$\begin{array}{rl} x, y \in & \mathsf{Variables} ::= \mathrm{x}_j \\ t, u \in & \overline{\lambda}'\text{-}\mathsf{Terms} ::= (x\ l) \mid (\lambda x.t) \mid \langle t_1, t_2\rangle \mid \mathrm{inj}_i(t) \\ l \in & \mathsf{Argument\text{-}Lists} ::= [\,] \mid [\langle (x_1)t_1 | (x_2)t_2\rangle] \mid [\langle x, y\rangle t] \mid [t :: l] \mid [\pi_i :: l] \end{array}$$

As usual, $[\,]$ is the empty list of arguments, and $[t :: l]$ is the list resulting from the addition of t at the beginning of l. We abbreviate $(x\ [\,])$ by x.

Solely to aid the reader's understanding of the meaning of $\overline{\lambda}'$-terms, we will relate them to terms of the λ-calculus extended with pairs and tagged variants. We define the extended λ-terms by this pseudo-grammar where $i \in \{1,2\}$:

$$\begin{array}{c} \hat{t} \in \lambda\text{-}\mathsf{Terms} ::= x \mid \lambda x.\hat{t} \mid \hat{t_1}\,\hat{t_2} \mid \langle \hat{t_1}, \hat{t_2}\rangle \mid \mathrm{inj}_i(\hat{t}) \mid \pi_i(\hat{t}) \mid \mathrm{let}\ x, y = \hat{t}\ \mathrm{in}\ \hat{u} \mid \\ \mathrm{case}\ \hat{t}\ \mathrm{of}\ \mathrm{inj}_1(x) \Rightarrow \hat{t_1}, \mathrm{inj}_2(x) \Rightarrow \hat{t_2} \end{array}$$

Now we translate $\overline{\lambda}'$-terms into extended λ-terms:

$$\begin{array}{ll} (x\ l)^* = \varphi(x, l) & \varphi(\hat{t}, [\,]) = \hat{t} \\ (\lambda x.t)^* = \lambda x.t^* & \varphi(\hat{t}, [u :: l]) = \varphi(\hat{t}\,u^*, l) \\ \langle t_1, t_2\rangle^* = \langle t_1^*, t_2^*\rangle & \varphi(\hat{t}, [\pi_i :: l]) = \varphi(\pi_i(\hat{t}), l) \\ & \varphi(\hat{t}, [\langle x, y\rangle u]) = \mathrm{let}\ x, y = \hat{t}\ \mathrm{in}\ \hat{u} \\ (\mathrm{inj}_i(t))^* = \mathrm{inj}_i(t^*) & \varphi(\hat{t}, [\langle (x_1)t_1 | (x_2) | t_2\rangle]) = \mathrm{case}\ \hat{t}\ \mathrm{of}\ \mathrm{inj}_1(x) \Rightarrow t_1^*, \mathrm{inj}_2(x) \Rightarrow t_2^* \end{array}$$

Let a *named environment* be a partial function from variables to formulas, and let Σ range over named-environments.

The rules of $\mathsf{LJT}^{\mathsf{Enum}}$ with the corresponding proof terms are given in Fig. 4. We call $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Term}}$ this set of rules.

Formulas in the goal are associated to a $\overline{\lambda}'$-expression. By construction, goals of rules in which the stoup is empty are $\overline{\lambda}'$-terms while those in which the stoup is full are lists of arguments waiting to be applied. Formulas which are in the stoup are not associated to a $\overline{\lambda}'$-expression, as is indicated by the notation ". : A".

Applicative contexts formation rules

$$\frac{}{\Sigma;.:X\vdash[\,]:X}\ \textsc{Ax}$$

$$\frac{\Sigma;\bullet\vdash u:A\qquad\Sigma;.:B\vdash l:C}{\Sigma;.:A\to B\vdash[u::l]:C}\ \textsc{Imp}_{\textsc{L}}$$

$$\frac{\Sigma;.:A_i\vdash l:B}{\Sigma;.:A_1\wedge A_2\vdash[\pi_i::l]:B}\ \textsc{And}_{\textsc{L}_i}$$

$$\frac{\Sigma,x:A;\bullet\vdash t:C\qquad\Sigma,y:B;\bullet\vdash u:C}{\Sigma;.:A\vee B\vdash[\langle(x)t|(y)u\rangle]:C}\ \textsc{Or}_{\textsc{L}}$$

Terms formation rules

$$\frac{\Sigma,x:A;.:A\vdash l:B}{\Sigma,x:A;\bullet\vdash(x\ l):B}\ \textsc{Cont}(x:A)$$

$$\frac{\Sigma,x:A;\bullet\vdash u:B}{\Sigma;\bullet\vdash\lambda x.u:A\to B}\ \textsc{Imp}_{\textsc{R}}$$

$$\frac{\Sigma;\bullet\vdash t:A\qquad\Sigma;\bullet\vdash u:B}{\Sigma;\bullet\vdash\langle t,u\rangle:A\wedge B}\ \textsc{And}_{\textsc{R}}$$

$$\frac{\Sigma;\bullet\vdash u:A_i}{\Sigma;\bullet\vdash\mathrm{inj}_i(u):A_1\vee A_2}\ \textsc{Or}_{\textsc{R}_i}$$

Fig. 4. Proof terms for the rules of $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Term}}(i\in\{1,2\})$

7.2 Number of different proof terms

Given a sequent s, there are strong ties between $\mathsf{C}_{\mathsf{MSet}}(s)$ and the number of different $\overline{\lambda}$-terms up to α-conversion which can be built from the proofs of s. In fact, the only source of difference is that $\mathsf{C}_{\mathsf{MSet}}(s)$ does not capture multiple uses of Cont on propositions which occur multiple times in the context, with different variable names.

From there, it is easy to devise a proof counting and enumerating algorithm for $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Term}}$: in $\mathsf{G}_{\mathsf{MSet}}$, just duplicate n times the edge between s and $(s,\ \textsc{Cont}(A))$ if A appears n times in the environment of s. All the results and theorems applicable to $\mathsf{G}_{\mathsf{MSet}}$ remain true with that modification. As a result, proof enumeration is no more difficult in $\mathsf{LJT}^{\mathsf{Enum}}_{\mathsf{Term}}$ than in $\mathsf{LJT}^{\mathsf{Enum}}$.

8 Related work

Dyckhoff and Pinto propose a confluent rewriting relation $\prec$ on the structure of cut-free proofs in LJ [8]. The normal forms of the proofs in LJ w.r.t. to $\prec$ are in 1-1 correspondence with normal natural deductions in NJ. That solution would not have been suitable for our purpose however, because we could easily have ended up finding an important number of proofs in LJ which would all have corresponded to the same normal proof in NJ.

Howe proposes two mechanisms to efficiently add an history to a sequent proof in LJT, in order to avoid loops in the proof [14]. One of these mechanisms has been added to our implementation of Count.

Pinto presents a mechanism to define names for proof-witnesses of formulae and thus to use Gentzen's cut-rule in logic programming [15]. Because using the cut-rule can make some proofs exponentially shorter, it should be possible to discover terms which are much more efficient from a computational standpoint

than those we can generate using a cut-free calculus. Devising an exhaustive term enumeration procedure for such a calculus would be an interesting task.

Ben-Yelles [3], Hindley [13], Zaionc [17], Broda and Damas [4] propose various algorithms to solve the problem of type inhabitation in the simply typed λ-calculus. Zaionc's approach is somewhat similar to our own, using fixpoints on polynomials. Broda and Damas propose a tool for studying inhabitation of simple types. In all four cases only simple types are considered, which makes those algorithms harder to use for "real-world" program synthesis.

9 Conclusion

9.1 Summary of contributions

We have presented COUNT, a proof counting algorithm for the $\mathsf{LJT}^{\mathsf{Enum}}$ calculus of intuitionistic logic. The idea is reusable for any calculus in which the environment of assumptions only increases (e.g., Gentzen's LJ). Using COUNT and the Curry-Howard correspondence, we have implemented an algorithm which effectively builds all the possible program fragments of a given typing.

We believe our approach to proof counting and enumeration is the first that has the following properties. First, we use the easier solution for assumption *sets* to build a more efficient solution for *multisets*, which is closer to our motivating goal of term enumeration. Second, our method works directly on logical-deduction style sequent derivations as normally used in proof search (i.e., L-systems with left-introduction rules instead of right-elimination rules), while earlier approaches instead count λ-terms in normal forms. Third, our method uses a graph representation of all proofs which seems essential for practicality.

9.2 Future work

Let us mention some promising ways to extend the expressiveness of our program fragments synthesizer. First, to better handle ML languages, adding some support for polymorphism would be useful; but this will break the syntax-directed property of the calculus, and probably the finiteness of $\mathsf{G}_{\mathsf{Set}}(s)$.

Ideally, we would also support full algebraic datatypes. We partially achieve this goal in that the method we present handles parametric types (e.g. the type constructor `list` as used in the type `int list` in Standard ML), provided the environment has functions to build and use them.

Furthermore, the addition of fully general sum types to model inductive datatypes, as well as of recursion, could also be interesting. This could be done for example using recursive propositions. However, a potential pitfall to avoid is generating "dead code" or predictably non-terminating functions.

Finally, while theoretically sound, the $\mathrm{OR_L}$ rule generates a huge number of λ-term which are extensionally equal. It is possible to rule out the less inefficient ones after they have been produced, but we are also investigating the possibility of pruning them during an earlier phase of the search.

References

[1] M. V. Aponte, R. Di Cosmo, C. Dubois, B. Yakobowski. Signature subtyping modulo type isomorphisms. In preparation, 2004.

[2] V. Balat, O. Danvy. Memoization in type-directed partial evaluation. In *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GCSE/SAIG)*, vol. 2487 of *LNCS*, Pittsburgh, PA, USA, 2002. ACM, ACM Press.

[3] C.-B. Ben-Yelles. *Type-assignment in the lambda-calculus; syntax and semantics*. PhD thesis, Mathematics Dept., University of Wales Swansea, UK, 1979.

[4] S. Broda, L. Damas. On the structure of normal λ-terms having a certain type. In *7th Workshop on Logic, Language, Information and Computation (WoLLIC 2000)*, Brazil, 2000.

[5] R. Di Cosmo, D. Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In A. Lingas, ed., *Proc. 20st Int'l Coll. Automata, Languages, and Programming*, vol. 700 of *LNCS*, 1993.

[6] R. Dyckhoff. Proof search in constructive logics. In *Logic Colloquium '97*, 1998.

[7] R. Dyckhoff, L. Pinto. Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia Logica*, 60(1), 1998.

[8] R. Dyckhoff, L. Pinto. Permutability of proofs in intuitionistic sequent calculi. *Theoret. Comput. Sci.*, 212(1–2), 1999.

[9] C. Haack. *Foundations for a tool for the automatic adaptation of software components based on semantic specifications*. PhD thesis, Kansas State University, 2001.

[10] C. Haack, B. Howard, A. Stoughton, J. B. Wells. Fully automatic adaptation of software components based on semantic specifications. In *Algebraic Methodology & Softw. Tech., 9th Int'l Conf., AMAST 2002, Proc.*, vol. 2422 of *LNCS*. Springer-Verlag, 2002.

[11] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Conf. Computer Science Logic*, vol. 933 of *LNCS*. Springer-Verlag, 1994.

[12] H. Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. Available at `http://coq.inria.fr/~herbelin/publis-eng.html`, 1994.

[13] J. R. Hindley. *Basic Simple Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[14] J. M. Howe. *Proof Search Issues In Some Non-Classical Logics*. PhD thesis, University of St Andrews, 1998.

[15] L. Pinto. Cut formulae and logic programming. In R. Dyckhoff, ed., *Extensions of Logic Programming: Proc. of the 4th International Workshop ELP'93*. Springer-Verlag, 1994.

[16] L. Pinto, R. Dyckhoff. Sequent calculi for the normal terms of the $\lambda\Pi$ and $\lambda\Pi\Sigma$ calculi. In D. Galmiche, ed., *Electronic Notes in Theoretical Computer Science*, vol. 17. Elsevier, 2000.

[17] M. Zaionc. Fixpoint technique for counting terms in typed lambda calculus. Technical Report 95-20, State University of New York, 1995.

Appendix

A Differences between LJT and LJT$^{\mathsf{Enum}}$

In this section, we explore the special requirements of a term enumeration calculus and justify the differences between LJT and LJT$^{\mathsf{Enum}}$. There are two differences between LJT and LJT$^{\mathsf{Enum}}$, which we study separately. Both have been introduced to avoid generating inefficient duplicates of other $\overline{\lambda}'$-terms.

A.1 Weakening of the Ax rule

We recall the Ax rule for LJT and LJT$^{\mathsf{Enum}}$:

$$\text{In LJT}^{\mathsf{Enum}}\text{: } \frac{}{\Gamma; X \vdash X}\ \text{Ax} \qquad \text{In LJT: } \frac{}{\Gamma; A \vdash A}\ \text{Ax}$$

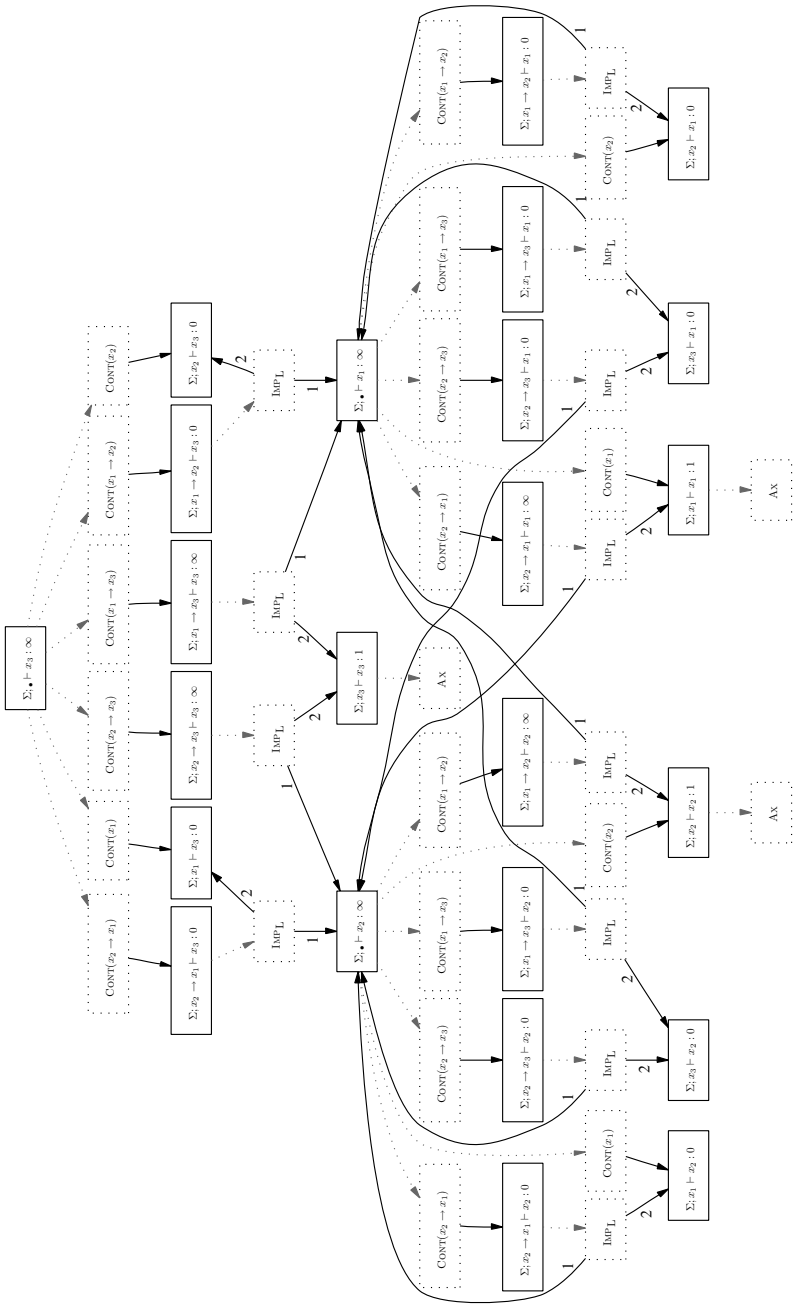

Fig. 5. $\mathsf{G}_{\mathsf{Set}}(\Sigma; \bullet \vdash x_3) = \mathsf{G}_{\mathsf{MSet}}(\Sigma; \bullet \vdash x_3)$ with $\Sigma = \{x_1, x_2, x_1 \to x_2, x_2 \to x_1, x_1 \to x_3, x_2 \to x_3\}$

Using LJT, we get 2 proofs for the sequent $A \to B; \vdash A \to B$: the first one uses Cont then Ax to conclude immediately, while the second one starts by decomposing $A \to B$ using Imp_R. If we call f the proposition of type $A \to B$ in the environment, the two proof terms we obtain are $(f\ [\,])$ and $\lambda x.(f\ [x])$.

The second expression can be trivially obtained by η-expanding the first one: we get two proof terms which are different by definition of the calculus, but which return the same result when applied to the same argument. By restricting the use of the Ax rule to basic propositions only, we forbid the first proof. As already noted by Dyckhoff and Pinto [6] this is equivalent to generating only terms in η-long normal form for implications. Similarly, conjunctions and disjunctions cannot be used "as is" and are first deconstructed, then reconstructed if needed.

Unfortunately, the restricted rule chooses the proof terms which incur a performance penalty: deconstructing then reconstructing a pair has a cost at execution time. Yet there is no easy way to get only the short forms. We can instead rely on external tools to simplify the $\overline{\lambda}'$-terms obtained. Rewriting systems for ML with product and sum types have already been proposed [5]. Recent developments in type-directed partial evaluation also provide a way to detect expressions which are basically the identity function for a complicated type [2].

A.2 Suppression of the symmetric And_L rule

Unlike in LJT^Enum, there is in LJT a third rule for conjunctions. It is interpreted using the split operator for pairs: instead of taking the left or the right part of the pair using projections, it adds both parts to the environment simultaneously.

$$\frac{\Gamma, A, B; \bullet \vdash C}{\Gamma; A \wedge B \vdash C}\ \text{And}_\text{L} \qquad \frac{\Gamma, x : A, y : B; \bullet \vdash u : C}{\Gamma; . : A \wedge B \vdash [\langle x, y\rangle u] : C}\ \text{And}_\text{L}$$

The rules And_L and $\text{And}_{\text{L}_\text{i}}$ are redundant: the use of the rules $\text{And}_{\text{L}_\text{i}}$ can be replaced by an application of And_L followed by Cont to put in the stoup the element of the pair which must be used. Replacing And_L by $\text{And}_{\text{L}_\text{i}}$ is a bit more difficult. The use of And_L must be pushed towards the leaves of the proof tree, until the moment one of the elements of the pairs is really used, i.e., put in the stoup. If both are used, the pair must be constructed twice (which is always possible because the environment never decreases in LJT), and decomposed twice. If no element is used, the application of And_L can simply be removed.

In light of this analysis, it is clear that proofs using And_L are always shorter than proofs using $\text{And}_{\text{L}_\text{i}}$. In fact, some are exponentially shorter. On the other hand, And_L empties the stoup without modifying the goal, and in some cases the content of the stoup is not used afterwards in the proof.

Consider for example the sequent $v : A \wedge B; \vdash X \to X$. Using And_L we get the following (infinite) sequence of $\overline{\lambda}'$-terms: $\lambda x.x$, $(v\ [\langle v_1, v_2\rangle\ \lambda x.x])$, $\lambda x.(v\ [\langle v_1, v_2\rangle\ x])$, $(v\ [\langle v_1, v_2\rangle\ (v\ [\langle v_3, v_4\rangle\ \lambda x.x])])$, and so on, which are nothing more than the identity. Because all these terms except $\lambda x.x$ are useless to us, we chose to remove And_L from LJT^Enum.

A.3 Remaining inefficiencies

In spite of the restrictions in $\mathsf{LJT}^{\mathsf{Enum}}$ (compared with LJT), some inefficiencies remain. One of them is the possibility of decomposing disjunctions more than once in a branch of the proof tree. This produces terms of the form $(x\,[\langle(y_1)(x\,[\langle(y_2)t_1|(y_3)t_2\rangle])|(y_2)x\rangle])$.

Another related source of inefficiency is the possibility of decomposing a disjunction, and then doing the same thing in both branches. One example of this is the term $(x\,[\langle(y_1)x|(y_2)x\rangle])$ which is really the identity.