The Formula-Tree Proof Method

Sabine Broda, Luís Damas

Technical Report Series: DCC-04-1

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150-180 Porto, Portugal

Tel: +351+22+6078830 – Fax: +351+22+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

The Formula-Tree Proof Method

Sabine Broda Luís Damas

DCC-FC & LIACC
Universidade do Porto

Abstract

We define an alternative representation for types in the simply typed λ-calculus or equivalently for formulas in the implicational fragment of intuitionist propositional logic, which gives us a better insight on the nature of a types structure and its relation to the structure of the set of its normal inhabitants. Based on this representation of a type φ, called its formula-tree, we define the notion of a valid proof-tree. Any such valid proof-tree represents a finite set of normal inhabitants of φ and every normal inhabitant corresponds to exactly one valid proof-tree, constructable with the primitive parts in the formula-tree of φ. Precise algorithms are given to establish this relation. This method can be used to easily recognize and prove results concerning inhabitation/provability.

1 Introduction

Typed versions of systems of λ-calculus have been studied since their appearance in the beginning of this century in [6] and [5], due to their importance to areas of mathematical logic and more recently to computer science. One of the fields with direct application is the area of functional programming (e.g. programming languages ML, Miranda or Haskell). The concept of functional programming relies on the representation of computable functions/algorithms by λ-terms and the execution of programs corresponds to the reduction of λ-terms. Type assignment to λ-terms provides thus for partial specification of the represented algorithms and can be used to show partial correctness of programs. Besides that, types are used to improve the efficiency of the compilation of functional programs. Almost all practical systems have the system $\mathbf{TA}_\lambda$ of simply typed λ-calculus at their basis. On the other hand, there exists a direct correspondence (cf. [11]), via Curry-Howard isomorphism, between $\mathbf{TA}_\lambda$ and the implicational fragment of intuitionistic propositional logic. In fact, a type φ can by assigned to some λ-term if and only if φ is a provable formula/theorem. Furthermore, every (normal) term to which φ can be assigned, i.e. every (normal) inhabitant of φ, represents a (normal) proof of φ in the natural deduction system for the implicational fragment of intuitionistic logic.

Thus, it is natural to seek for algorithms which, for a given type φ, - decide if φ has any normal inhabitant, - decide if the number of normal inhabitants of φ is finite or infinite, - or compute the set of all normal (or even principal normal) inhabitants of φ. Decision-procedures for intuitionistic propositional logic have been known for a long time (e.g. [8], [7]) and the corresponding decision problem, as well as the infiniteness problem of the set of normal inhabitants have been shown to be polynomial-space complete, respectively by Statman in [13] and Hirokawa in [10]. An algorithm, also described in [9], to count (which is the difficult part) and list the normal inhabitants of a type φ has been given by Ben-Yelles in [2] and a similar algorithm for principal normal inhabitants was first presented in [3]. Both these algorithms are based on the generation of terms whose depth (a measure on the structure of a λ-term in β-normal form) does not exceed some limit which can be computed from φ. This fact makes both algorithms extremely inefficient. On the other hand, it seems to be very unsatisfactory that the algorithms mentioned do not take into account any information about φ, besides the number of type variables and of their occurrences. In fact, as pointed out by Hindley in [9], when we ask for the number of normal inhabitants [1] of a type, the answer is often finite and interesting patterns show up which are still not completely understood. The situation is even worse for principal inhabitants, since we do not know any "non-exhaustive" method for computing a principal normal inhabitant of a type φ. The method presented in this report is an attempt towards a better understanding of the nature of the structure of a type and its relation to the structure of the set of its normal inhabitants.

1.1 The Formula-tree Proof Method

We assume familiarity with the basic notions in λ-calculus and use standard notation from [1] and [9]. Our notation differs from that in [1], since we denote type-variables (atoms) by "A,B,C,..."and arbitrary types by lower-case Greek letters. For type assignment we consider the system $\mathbf{TA}_\lambda$ of simply typed λ-calculus à la Curry (for an introduction see [9]) or [1], section 3.1).

A β-normal inhabitant M of a type τ is called a *long* normal inhabitant of τ iff every variable-occurrence z in M is followed by the longest sequence of arguments allowed by its type, i.e. iff each component with form $(zP_1 \dots P_n)$, $(n \geq 0)$, that is not in a function position has atomic type. The finite set of all terms obtained by η-reducing a λ-term M zero or more times is called the η-family of M and denoted by $\{M\}_\eta$. It has been shown (cf. [2], [9]) that the η-families of the long normal inhabitants of τ partition the set of normal inhabitants of τ into non-overlapping finite subsets, each η-family containing just one long member. Furthermore, Ben-Yelles (cf. [2], [9]) showed that every normal inhabitant of a type τ can be η-expanded to one unique (up to α-conversion) long normal inhabitant of τ. A simple expansion-algorithm can be found in [9].

In the following we describe the straight relation that exists between subterms and variables in a long normal inhabitant of a type φ and the subtypes of φ. It is well known that for every (long) normal inhabitant M of a type φ there is exactly one deduction in the system $\mathbf{TA}_\lambda$ that assigns the type φ to M. Thus, in the remaining we sometimes refer to the types that are assigned to variables and subterms of a normal inhabitant during this unique deduction as *their* types. We begin recalling the rather standard definition of polarities of occurrences of subtypes/subformulas.

Definition 1.1 An occurrence of a subtype in a type is defined as *positive* or *negative* as follows.

- φ occurs positively in φ;
- if an occurrence of a subtype is positive (negative) in φ, then it is negative (resp. positive) in $\varphi \to \tau$;
- if an occurrence of a subtype is positive (negative) in τ, then it is positive (resp. negative) in $\varphi \to \tau$.

The following properties of long normal inhabitants follow almost directly from the definitions and give us some insight on the relation between the structure of a type and the structure of their long normal inhabitants (an exhaustive analysis of this relationship can be found in [9], cf. subsection 8E - The structure of a nf-scheme).

[1] Note that the number of (not necessarily normal) inhabitants of a type is always zero or infinite.

Lemma 1.2 *Consider a long normal inhabitant M of a type φ and let P be a subterm of M.*

i. *If $P = \lambda x_1 \ldots x_n.N$, with $n \geq 1$ and such that N is no abstraction, then there is some positive occurrence of a subtype of the form $\tau = \alpha_1 \to \ldots \to \alpha_n \to A$ in φ and P is of type τ. Furthermore, $x_1, \ldots, x_n$ are respectively of type $\alpha_1, \ldots, \alpha_n$, which are negative occurrences of subtypes of φ, while N is of atomic type A.*

ii. *If $P = yM_1 \ldots M_n$, with $n \geq 0$ and such that $M_1, \ldots, M_n$ are all the arguments applied to y, then there is some negative occurrence of a subtype of the form $\tau = \alpha_1 \to \ldots \to \alpha_n \to A$ in φ, y has type τ and $P, M_1, \ldots, M_n$ have respectively types $A, \alpha_1, \ldots, \alpha_n$.* •

We conclude mainly that variables in abstraction sequences correspond to negative occurrences of subtypes and that subterms in argument position correspond to positive occurrences of subtypes. Also, every variable x with a type $\alpha_1 \to \ldots \to \alpha_n \to A$ occurs always with exactly n arguments in a long normal inhabitant, i.e. in an expression of the form $xP_1 \ldots P_n$ of type A and such that $P_1, \ldots, P_n$ are respectively of types $\alpha_1, \ldots, \alpha_n$. Moreover, if $\alpha_i = \beta_1 \to \ldots \to \beta_k \to B$, $i \in \{1, \ldots, n\}$, then P_i is of the form $\lambda y_1 \ldots y_k.N$ such that $y_1, \ldots, y_k$ have types $\beta_1, \ldots, \beta_k$ which are negative occurrences of subtypes, and N has type B.

The *Formula-Tree Proof Method* described in this report, was first and more informally introduced in [4][2]. In this method every type φ is split into primitive parts, and the primitive parts themselves form a tree-like structure, which is called the *formula-tree* of φ, and that defines some kind of hierarchy over the primitive parts of the formula. Here, every primitive part, except the one in the root of the formula-tree, represents together with the subtree rooted in it a subtype of φ that, during the construction of a long normal inhabitant of φ, may be assigned to variables[3]. The formula-tree of φ, `tree`(φ), can then be used to build *proof-trees* for φ, each proof-tree being a representation of a finite non-empty set of long normal inhabitants of φ. Since φ may have none, a finite or an infinite number of long normal inhabitants it may consequently be possible to build none, a finite or even an infinite number of different proof-trees for φ.

Primitive parts (frequently represented inside of boxes) are items of either one of the following forms (P1), (P2) or (P3), where $A, B, B_1, \ldots, B_n, C$ denote type-variables:

(P1): $\boxed{\begin{array}{c} | \\ A \end{array}}$ (P2): $\boxed{\begin{array}{c} B \\ / \quad \backslash \\ B_1 \ \ldots \ B_n \end{array}}$ $(n \geq 1)$ (P3): $\boxed{\begin{array}{c} C \\ | \end{array}}$

Here, $A, B_1, \ldots, B_n$ are called the *tail-variables* of the respective primitive part, while B and C are *head-variables*. The *arity* of a primitive part is the number of its tail-variables.

Definition 1.3 A formula-tree is a tree-like structure with primitive parts as nodes, but such that subtrees descend from the branches (or tail-variables) of primitive parts rather than from the whole primitive parts as nodes. Then, such a structure is called a *formula-tree* iff

- the root of the formula-tree is the only primitive part of form (P1);
- every node of form (P2) or (P3) in the formula-tree descends from a tail-variable in another primitive part;
- every (labeled) primitive part occurs only once in a formula-tree.

---

[2] A short presentation of the method together with an implementation as a Java applet can be found at `http://www.ncc.up.pt/~sbb/FTLab/ftlab`

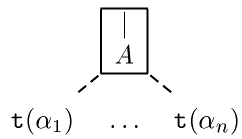
[3] In [9] these subtypes are also called negative subpremises.

The following algorithm computes the formula-tree $\texttt{tree}(\varphi)$ of a type φ. We use dashed lines for the edges of the formula-tree in order to distinguish them from the edges in the primitive parts (nodes) of the formula-tree. Furthermore, we will use the expression *branch* only when referring to edges in primitive parts, but not when referring to (dashed) edges of formula-trees.

Formula-tree Algorithm 1.4
Input: A type $\varphi = \alpha_1 \to \ldots \to \alpha_n \to A$, where A is an atom and $n \geq 0$.
Output: φ's formula-tree $\texttt{tree}(\varphi)$.

The formula-tree $\texttt{tree}(\varphi)$ is given by the following.

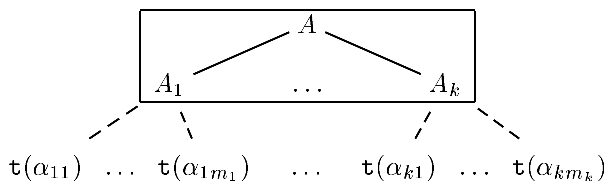
- If $n = 0$, i.e. $\varphi \equiv A$, then $\texttt{tree}(\varphi) = \boxed{\begin{array}{c} | \\ A \end{array}}$.

- If $n \geq 1$, then $\texttt{tree}(\varphi) =$ 

  where $\texttt{t}(A) = \boxed{\begin{array}{c} A \\ | \end{array}}$

  and for $k \geq 1$ and $m_1, \ldots, m_k \geq 0$ we recursively define

  $\texttt{t}((\alpha_{11} \to \ldots \to \alpha_{1m_1} \to A_1) \to \ldots \to (\alpha_{k1} \to \ldots \to \alpha_{km_k} \to A_k) \to A) =$

  Furthermore, for $1 \leq j \leq k$, we call the primitive parts in the top of $\texttt{t}(\alpha_{j1}), \ldots, \texttt{t}(\alpha_{jm_j})$ the *descendents* of primitive part $\boxed{\begin{array}{c} A \\ / \quad \backslash \\ A_1 \ \ldots \ A_k \end{array}}$ at branch j (or tail-variable A_j). Note, that in case $m_j = 0$, branch j, i.e. tail-variable A_j, has no children/descendents.

We assume that all primitive parts introduced above are given distinct labels, thus ensuring that each primitive part occurs only once as required in the definition of a formula-tree.

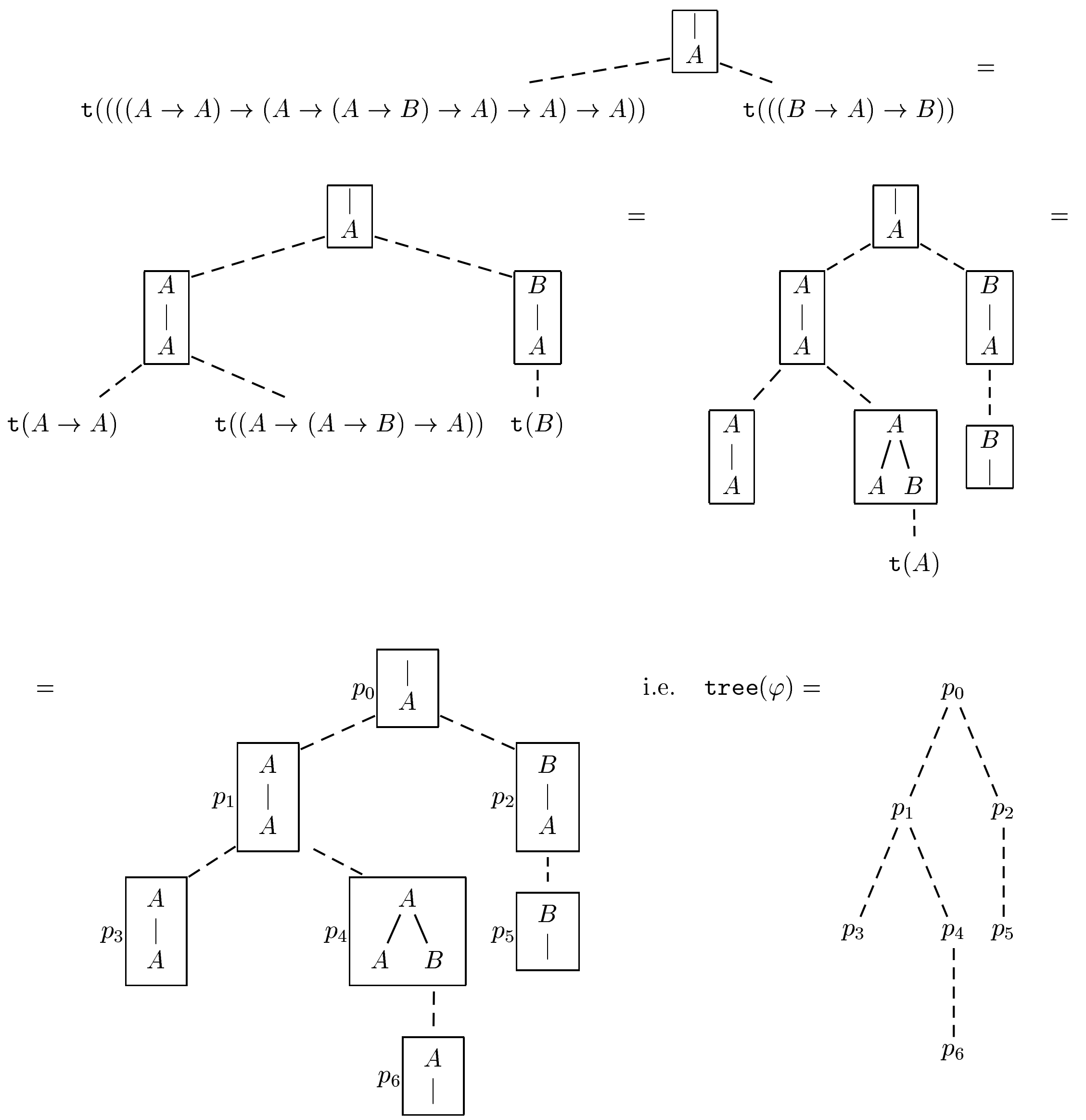
In the following we exemplify the formula-tree algorithm as well as the notion of *descendents of a primitive part at a branch (tail-variable).*

Example 1.5 The formula

$$\varphi = (((A \to A) \to (A \to (A \to B) \to A) \to A) \to A) \to ((B \to A) \to B) \to A$$

has formula-tree

$$\texttt{tree}(\varphi) = \texttt{tree}((((A \to A) \to (A \to (A \to B) \to A) \to A) \to A) \to ((B \to A) \to B) \to A) =$$



with primitive parts $p_0 = \boxed{\begin{array}{c} | \\ A \end{array}}$, $p_1 = \boxed{\begin{array}{c} A \\ | \\ A \end{array}}$, $p_2 = \boxed{\begin{array}{c} B \\ | \\ A \end{array}}$, $p_3 = \boxed{\begin{array}{c} A \\ | \\ A \end{array}}$, $p_4 = \boxed{\begin{array}{c} A \\ / \ \backslash \\ A \quad B \end{array}}$, $p_5 = \boxed{\begin{array}{c} B \\ | \end{array}}$ and $p_6 = \boxed{\begin{array}{c} A \\ | \end{array}}$. In the formula-tree $\mathtt{tree}(\varphi)$, p_0 has descendents p_1 and p_2 at p_0's branch 1 (p_0's only branch), p_1 has descendents p_3 and p_4 at p_1's branch 1 (p_1's only branch), p_2 has descendent p_5 at its only branch 1, p_4 has descendent p_6 at branch 2 and no descendent at branch 1, and finally p_3, p_5 and p_6 have no descendents at all.

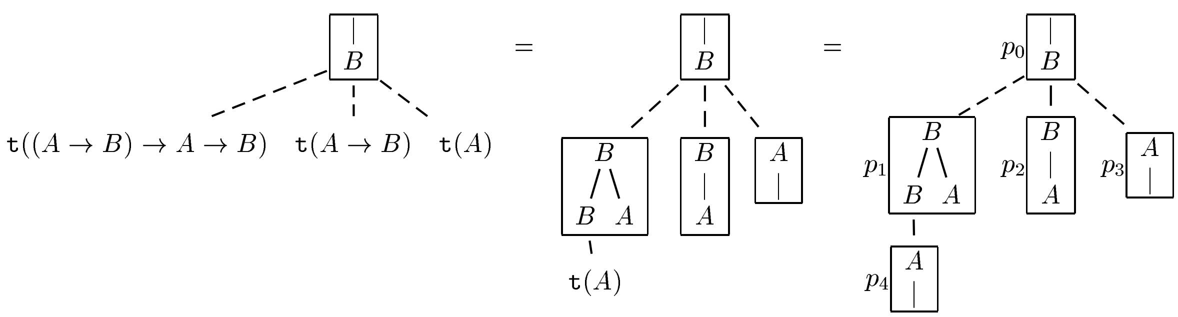
It is easy to see that for every negative occurrence of a subtype α of a type φ there will be a subtree $\mathtt{t}(\alpha)$ of $\mathtt{tree}(\varphi)$, suggesting the following relation between subtrees and the types of variables in long normal inhabitants (cf. Lemma 1.2): given a long normal inhabitant M of a type φ and a variable x in M of type α there is exactly one subtree $\mathtt{t}(\alpha)$ of $\mathtt{tree}(\varphi)$ corresponding to x. Any such subtree can be identified by the primitive part in its top position. So, from now on we call this *the primitive part corresponding to the variable x in M*. Note that, while every variable in M has exactly one corresponding primitive part in $\mathtt{tree}(\varphi)$, given a primitive part p in $\mathtt{tree}(\varphi)$ there may be more than one variable in M for which p is the corresponding primitive part, or there may even be no such variable. The result below follows from the definition of *corresponding primitive part* and from the observations made after Lemma 1.2.

Lemma 1.6 *Let M be a long normal inhabitant of a type φ with formula-tree $\mathtt{tree}(\varphi)$, consider any variable x in M and let p be the primitive part in $\mathtt{tree}(\varphi)$ corresponding to x. If p is of arity $k \geq 0$, then x occurs always with exactly k arguments in M. Furthermore, for $k \geq 1$, if $i \in \{1, \ldots, k\}$ and p has descendents $p_1, \ldots, p_m$ at branch/tail-variable i in $\mathtt{tree}(\varphi)$, then the ith argument of x is of the form $\lambda y_1 \ldots y_m.N$ and the primitive parts in $\mathtt{tree}(\varphi)$ corresponding to $y_1, \ldots, y_m$ are respectively $p_1, \ldots, p_m$.* •

Furthermore, one can easily define an inverse algorithm, which given a formula-tree $\mathtt{FT}$ computes the unique type φ such that $\mathtt{tree}(\varphi) = \mathtt{FT}$. In the following example we give some intuition on the meaning of primitive parts and the hierarchy defined on them.

Example 1.7 The formula $\varphi = ((A \to B) \to A \to B) \to (A \to B) \to A \to B$ has formula-tree

$\mathtt{tree}(\varphi) = \mathtt{tree}(((A \to B) \to A \to B) \to (A \to B) \to A \to B) =$



i.e. $\mathtt{tree}(\varphi) = \begin{array}{ccc} & p_0 & \\ p_1 & p_2 & p_3 \\ p_4 & & \end{array}$ with primitive parts $p_0 = \boxed{\begin{array}{c} | \\ B \end{array}}$, $p_1 = \boxed{\begin{array}{c} B \\ / \ \backslash \\ B \quad A \end{array}}$, $p_2 = \boxed{\begin{array}{c} B \\ | \\ A \end{array}}$,

$p_3 = \boxed{\begin{array}{c} A \\ | \end{array}}$ and $p_4 = \boxed{\begin{array}{c} A \\ | \end{array}}$. Note that part p_4 connects to the branch 1 of part p_1 with tail variable B.

Then we get the following information from this representation: during the construction of a long normal inhabitant of φ there may appear variables of type $(A \to B) \to A \to B$, of type $A \to B$ or of type A, which are the negative subtypes of φ, cf. Lemma 1.2. These are represented respectively by the subtrees $\mathtt{t}((A \to B) \to A \to B)$, $\mathtt{t}(A \to B)$ and $\mathtt{t}(A)$ (the latter occurs twice). As such, using a variable x of type $(A \to B) \to A \to B$, corresponding to the subtree $\mathtt{t}((A \to B) \to A \to B)$ with primitive part $p_1 = \boxed{\begin{array}{c} B \\ / \ \backslash \\ B \quad A \end{array}}$ at its top, allows us to construct a term of type B — the head-variable of p_1. Furthermore, in order to complete the construction of this term, two other terms of type B and of type A respectively — the tail-variables of p_1 — have to be constructed. Also x will appear with two — the arity of p_1 — arguments, and for the construction of the first there will appear a new variable of type A represented by $\mathtt{t}(A)$, i.e. corresponding to p_4 — the descendent of the first branch of p_1. Actually, all this parallels the fact, cf. Lemma 1.2, that every long normal term beginning with a variable x of type $(A \to B) \to A \to B$ is of the form $x(\lambda y.M)N$ of type B, where y, M and N are respectively of type A, B and A, and finally in M there may appear a new variable y of type A.

The previous discussion illustrates that we can associate to each variable in a long normal inhabitant M of a type φ a subtree $\mathtt{t}(\alpha_i)$ of $\mathtt{tree}(\varphi)$ with primitive part p_i at its top (the corresponding primitive part). Now, consider M' obtained from M by replacing each variable in M with the name of the corresponding primitive part and by erasing abstractions. For example, for the long normal inhabitant $M = \lambda xyz.x(\lambda w.yw)z$ of φ as above we obtain $M' = p_1(p_2p_4)p_3$, which represented as a tree gives, after inserting a top level node p_0, the following tree of primitive parts

$$\begin{array}{ccc} & p_0 & \\ & \| & \\ & p_1 & \\ /\!\!/ & & \backslash\!\!\backslash \\ p_2 & & p_3 \\ \| & & \\ p_4 & & \end{array}$$

It turns out that one can characterise precisely the trees of primitive parts that correspond to long normal inhabitants of a type and which will be called *valid proof-trees*. We now describe the set of rules that allows us to build proof-trees and then give two conditions to be satisfied by a proof-tree in order for it to be valid. In the remaining we will often use the notation $\overline{o}$ to refer to a specific occurrence of an object o.

Definition 1.8 [**Proof-tree Structure**] A *proof-tree* for a type φ is a tree-like structures whose nodes are primitive parts in $\mathtt{pp}(\varphi) = \{p_0, \ldots, p_n\}$, the set of (labels of the) primitive parts in the formula-tree of φ and where p_0 denotes the primitive part in the root of $\mathtt{tree}(\varphi)$ (thus the only primitive part of form (P1) in $\mathtt{pp}(\varphi)$). In the following we shall use the notation $\left|\begin{array}{c}\vdots\\ A\end{array}\right|$ to designate a proof-tree that has in a leaf a primitive part with a tail-variable A.

Then, the set of proof-trees for φ is given by the following:

- If $p_0 = \boxed{\begin{array}{c}|\\ A\end{array}} \in \mathtt{pp}(\varphi)$, then $\boxed{\begin{array}{c}|\\ A\end{array}}$ is a proof-tree for φ;
- if $\left|\begin{array}{c}\vdots\\ B\end{array}\right|$ is a proof-tree for φ and $\boxed{\begin{array}{ccc} & B & \\ / & & \backslash \\ B_1 & \ldots & B_k\end{array}} \in \mathtt{pp}(\varphi)$, where $k \geq 1$, then $\begin{array}{c}\left|\begin{array}{c}\vdots\\ B\end{array}\right| \\ \| \\ \boxed{\begin{array}{ccc} & B & \\ / & & \backslash \\ B_1 & \ldots & B_k\end{array}}\end{array}$

  is a proof-tree for φ;
- if $\left|\begin{array}{c}\vdots\\ B\end{array}\right|$ is a proof-tree for φ and $\boxed{\begin{array}{c}B\\ |\end{array}} \in \mathtt{pp}(\varphi)$, then $\begin{array}{c}\left|\begin{array}{c}\vdots\\ B\end{array}\right| \\ \| \\ \boxed{\begin{array}{c}B\\ |\end{array}}\end{array}$ is a proof-tree for φ.

Definition 1.9 [**Valid Proof-trees**] Given a proof-tree $\mathtt{PT}$ for a type φ, we call it a *valid proof-tree* for φ iff

- the number of subtrees rooted in any node/primitive part in $\mathtt{PT}$ equals the arity of that primitive part;

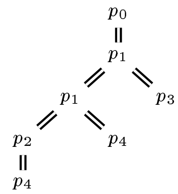
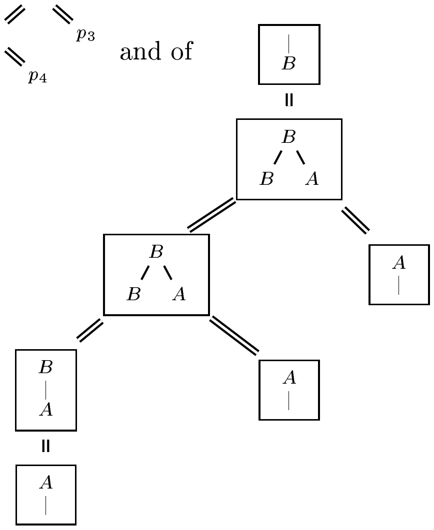
- if $p' = \boxed{\begin{array}{ccc} & B & \\ / & & \backslash \\ B_1 & \dots & B_k \end{array}}$, with $k \geq 1$ is a primitive part in the formula-tree $\texttt{tree}(\varphi)$, and p is a descendent of p' in $\texttt{tree}(\varphi)$ at branch i, with $1 \leq i \leq k$, then above every occurrence $\overline{p}$ of p in $\texttt{PT}$ there is at least one occurrence $\overline{p}'$ of p' in $\texttt{PT}$, such that the occurrence $\overline{p}$ occurs in the subtree of $\texttt{PT}$ rooted in the i'th branch of $\overline{p}'$.

This last item corresponds to the hierarchy of $\texttt{tree}(\varphi)$ being imposed on the construction of a valid proof-tree and will be illustrated by the following example. From now on, we will usually, and for the sake of saving space, represent *valid proof-trees* without drawing edges between primitive parts explicitly. As such, the tree of primitive parts at the end of example 1.7 will be represented by $\begin{array}{ll} & p_0 \\ & p_1 \\ p_2 & p_3 \\ p_4 & \end{array}$. Due to the first condition in the definition of a valid proof-tree (Definition 1.9) there will be no ambiguity regarding descendents/parents in the tree. Also, for the sake of saving space, we will *unify* the head variable and tail variables of two parts that are linked in a proof-tree.

Example 1.10 A valid proof-tree for the formula φ in example 1.7 is $\begin{array}{c} p_0 \\ \| \\ p_2 \\ \| \\ p_3 \end{array}$ i.e. $\begin{array}{c} \boxed{\begin{array}{c} | \\ B \end{array}} \\ \| \\ \boxed{\begin{array}{c} B \\ | \\ A \end{array}} \\ \| \\ \boxed{\begin{array}{c} A \\ | \end{array}} \end{array}$. Using the compact representation this proof-tree will be represented by $\begin{array}{c} p_0 \\ p_2 \\ p_3 \end{array}$ and has appearance $\begin{array}{c} | \\ B \\ | \\ A \\ | \end{array}$ after unifying/overlapping the linked variables. On the other hand, the tree $\begin{array}{c} p_0 \\ p_2 \\ p_4 \end{array}$ is not a valid proof-tree for φ (though it would have the same appearance), since it does not respect the hierarchy given by the formula-tree of φ, which requires that in a valid proof-tree all occurrences of p_4 occur in some subtree that is rooted in the left branch of an occurrence of p_1. Moreover, there are exactly three proof-trees with appearance

$$\begin{array}{ccc} & | & \\ & B & \\ / & & \backslash \\ B & & A \\ | & & | \\ A & & \\ | & & \end{array}, \quad \text{that are} \quad \begin{array}{ll} & p_0 \\ & p_1 \\ p_2 & p_3 \\ p_4 & \end{array}, \quad \begin{array}{ll} & p_0 \\ & p_1 \\ p_2 & p_3 \\ p_3 & \end{array} \quad \text{and} \quad \begin{array}{ll} & p_0 \\ & p_1 \\ p_2 & p_4 \\ p_4 & \end{array}.$$

Only the first two of these proof-trees are valid proof-trees, since the third does again not satisfy the second condition of definition 1.9.

Another valid proof-tree is $\begin{array}{lll} & & p_0 \\ & & p_1 \\ & p_1 & p_3 \\ p_2 & p_4 & \\ p_4 & & \end{array}$ with appearance $\begin{array}{ccccc} & & | & & \\ & & B & & \\ & / & & \backslash & \\ & B & & & A \\ / & & \backslash & & | \\ B & & A & & \\ | & & | & & \\ A & & & & \\ | & & & & \end{array}$. Note that these again are, respectively, the compact representations of  and of  .

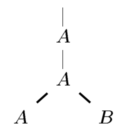
The following example will give some intuition on the fact that, given a formula φ, the search for a valid proof-tree, and consequently the search for normal inhabitants, is finite. A sketch of a search-algorithm is given after the example.

Example 1.11 Consider again the formula φ and its formula-tree $\mathtt{tree}(\varphi)$ from example 1.5. We will now attempt to build a valid proof-tree for this formula, applying the rules for constructing proof-trees, in such a way that none of the conditions in definition 1.9 is violated:

1. The only primitive part that can be used in the beginning is p_0, leading to the tree p_0 with appearance $\begin{array}{c}| \\ A\end{array}$. This is no valid proof-tree, since p_0 has arity 1 and consequently there has to be a subtree rooted in A.

2. Now, in order to build that subtree, applying one of the other two construction-rules for proof-trees, we have to use a primitive part with head-variable A. The primitive parts in the formula-tree with head-variable A are p_1, p_3, p_4 and p_6. But, actually the only primitive part that can be used without violating the second condition in definition 1.9 is p_1. Being a descendent of p_0 at branch 1, this primitive part, as well as p_2, has been made available by the use of p_0. The resulting proof-tree is $\begin{array}{c}p_0 \\ p_1\end{array}$ with appearance $\begin{array}{c}| \\ A \\ | \\ A\end{array}$, that is still not complete.

3. Again a subtree rooted in type-variable A has to be constructed, but now there are two more primitive parts that may be used. In fact, p_3 and p_4 (the descendents of p_1 at branch 1) have been made available by the use of p_1, and so one may now choose between p_1, p_3 or p_4 (p_2 is available, but its head-variable does not fit). Using p_1 again would lead to the still incomplete proof-tree $\begin{array}{c}p_0 \\ p_1 \\ p_1\end{array}$ with appearance $\begin{array}{c}| \\ A \\ | \\ A \\ | \\ A\end{array}$. Again a subtree rooted in A has to be constructed and the use of p_1 does not allow us to use any primitive part different from those available in the step before. The same will happen if we choose to use p_3 instead of p_1. In fact, p_3 has no descendents and consequently there will be no further primitive parts available, but the same as in the step before. Furthermore, p_3 has also tail-variable A and consequently we will still be attempting to construct a subtree rooted in A. Finally, we can use part p_4 which leads us

to the incomplete proof-tree $\begin{array}{c} p_0 \\ p_1 \\ p_4 \end{array}$ with appearance 

. Now, we have to construct two subtrees. One rooted in type-variable A and another rooted in type-variable B. Note, that p_4 has no descendent at branch 1 and the descendent p_6 at branch 2. Although, p_6 is now available for the construction of the subtree rooted in B, there is no new primitive part available for the construction of the subtree rooted in A. Consequently, again one of our tasks is the same as before and we conclude that there is no valid proof-tree for this formula.

Based on the argument used in the previous example one can define a search algorithm for a valid proof-tree as follows. Begin the construction of a valid proof-tree with p_0, the unique primitive part of form (P1) in the formula-tree. Proceed, building a complete subtree rooted at the tail-variable of p_0. During the construction of subtrees only use primitive parts that are allowed by the second condition in definition 1.9. Call these the available parts at a certain stage. During the construction, explore all possibilities, using all available primitive part with the matching head-variable. The search stops in one of two ways. With success, if a valid proof tree is obtained. With failure, whenever trying to build a complete subtree rooted in the same type-variable and with the same set of available primitive parts as in one of the steps before; this means that there is no need to construct proof-trees with depth greater than the number of different type-variables (in the type) times the number of primitive parts (in the formula-tree). Eventually, one of the two halting conditions will be obtained, since the number of atoms and primitive parts found in φ is finite.

In order to establish the relation between the long normal inhabitants of a type and its valid proof-trees we now describe an algorithm $\texttt{PT}(\_)$ that given a long inhabitant M of a type φ computes a valid proof-tree for $\texttt{PT}(M)$ for φ. The definition of this algorithm parallels the construction of a proof-tree from a long normal inhabitant in example 1.7. In the following algorithm we use the notion of erasing abstractions in a term, which consists of removing all prefixes λx in the term, obtaining as a result a purely applicative term.

Proof-tree Algorithm 1.12
Input: a type φ and M, a long normal inhabitant of φ.
Output: a valid proof-tree $\texttt{PT}(M)$ for φ.

Let M' be the term obtained from M by erasing abstractions in M and replacing each variable with the name of the corresponding primitive part in $\texttt{tree}(\varphi)$. Then, $\texttt{PT}(M)$ is the graphical representation of M' after inserting a top level node p_0 (the root-node of $\texttt{tree}(\varphi)$).

We now examine an "inverse" algorithm which computes, given a type φ and any valid proof-tree $\texttt{PT}$ for φ, a finite non-empty set $\texttt{Terms}(\texttt{PT})$ of long closed normal inhabitants of φ. The algorithm consists of three parts.

In the first two parts, the common structure of the lambda-terms that correspond to a certain valid proof-tree is recovered by a λ-term, that we call a *term-scheme*. These term-schemes resemble lambda-terms (possibly with bounded-variable clashes) in the simple type-system with the total discharge convention, or Prawitz-style natural deduction system [12], but actually they differ from them in the following way. While all free (distinct) variables in a subterm of a term in Prawitz' system have distinct types, distinct (free or bounded) variables in a subterm of a term-scheme may have the same type as long as they correspond to different primitive parts (or equivalently to different negative occurrences of the type as a subtype).

In the last part we compute the set of all terms from which a given term-scheme may be obtained, by identifying in these terms all variables, that correspond to the same primitive part in the formula-tree.

For example, for the term-scheme $\lambda xy.x(\lambda z.x(\lambda z.yzz))$ of type $((B \to A) \to A) \to (B \to B \to A) \to A$, the two bounded z's are given distinct names and all possibilities explored for the subterms in their scope, leading to the following set of terms

$$\{\lambda xy.x(\lambda z.x(\lambda z'.yzz)), \lambda xy.x(\lambda z.x(\lambda z'.yzz')), \lambda xy.x(\lambda z.x(\lambda z'.yz'z)), \lambda xy.x(\lambda z.x(\lambda z'.yz'z'))\}.$$

Terms Algorithm 1.13
Input: a type φ and a valid proof-tree `PT` for φ.
Output: a finite non-empty set `Terms(PT)` of long closed normal inhabitants of φ.

Let $p_0, \ldots, p_n$ be the primitive parts in `tree`(φ), where p_0 is the root-part and $n \geq 0$. The set `Terms(PT)` is constructed in the following way.

(a) Represent `PT` as an application using for $0 \leq i \leq n$ the variable name x_i instead of p_i. Then, for each variable-occurrence x_i in this application, such that the primitive part p_i has arity $k_i \geq 0$, insert a (possibly empty) abstraction sequence before each of its k_i arguments. Here, the variable names in an abstraction sequence $\lambda x_{j_1} \ldots x_{j_{l_j}}$ to be inserted before the jth argument, $1 \leq j \leq k_i$, correspond to the descendents $p_{j_1} \ldots p_{j_{l_j}}$ at branch j of p_i in `tree`(φ).

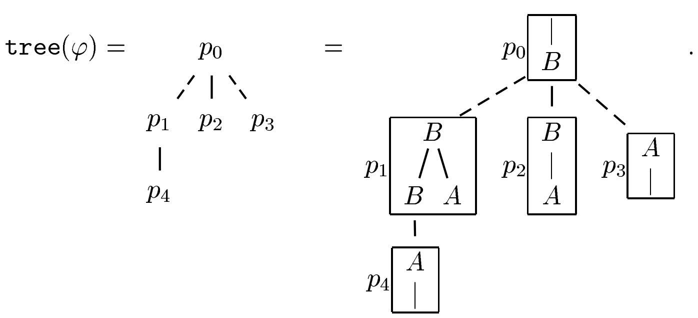
(b) Now, erase the variable x_0 at the top.

(c) Finally, for term-scheme T obtained in the previous step compute `Terms(PT)` $= T^{0\ldots 0}_{x_1\ldots x_n}$ defined below, where $x^0 = x$ for any variable x.

- $x_{i\,x_1\ldots x_i\ldots x_n}^{k_1\ldots 0\ldots k_n} = \{\overline{x}_i\}$;[4]
- $x_{i\,x_1\ldots x_i\ldots x_n}^{k_1\ldots k_i\ldots k_n} = \{x_i, x_i', x_i'', \ldots, x_i^{k_i-1}\}$, $\quad k_i \geq 1$;
- $(ST)^{k_1\ldots k_n}_{x_1\ldots x_n} = \{S_iT_j \mid S_i \in S^{k_1\ldots k_n}_{x_1\ldots x_n}, T_j \in T^{k_1\ldots k_n}_{x_1\ldots x_n}\}$;
- $(\lambda x_i.T)^{k_1\ldots k_n}_{x_1\ldots x_n} = \{\lambda x_i^{k_i}.T_i \mid T_i \in T^{k_1\ldots(k_i+1)\ldots k_n}_{x_1\ldots x_i\ldots x_n}\}$.

The following example illustrates the application of the algorithm.

Example 1.14 Consider again $\varphi = ((A \to B) \to A \to B) \to (A \to B) \to A \to B$ from example 1.7 with formula-tree



Let `PT` be the last valid proof-tree for φ in Example 1.10, i.e. `PT` =

$$\begin{array}{c} p_0 \\ p_1 \\ p_1\ p_3 \\ p_2\ p_4 \\ p_4 \end{array}$$

[4]This case corresponds to the renaming of free occurrences of variables in T.

In the first step of the Terms algorithm, PT is represented by the application $x_0(x_1(x_1(x_2x_4)x_4)x_3)$. Then, since p_0 has arity 1 and descendents p_1, p_2 and p_3 at branch 1, the abstraction sequence $\lambda x_1x_2x_3$ is inserted before the first argument of x_0, leading to $x_0(\lambda x_1x_2x_3.x_1(x_1(x_2x_4)x_4)x_3)$. As p_1 has arity 2, descendent p_4 at branch 1 and no descendent at branch 2, the abstraction sequence λx_4 has to be inserted before the first argument of (every occurrence of) x_1 and no (or the empty) abstraction sequence will be inserted before the second argument of (all occurrences of) x_1, leading to $x_0(\lambda x_1x_2x_3.x_1(\lambda x_4.x_1(\lambda x_4.x_2x_4)x_4)x_3)$. Neither p_2, p_3 or p_4 have descendents; thus no further abstraction sequence has to be inserted. After erasing x_0 at the top we obtain the term-scheme $T = \lambda x_1x_2x_3.x_1(\lambda x_4.x_1(\lambda x_4.x_2x_4)x_4)x_3$. Finally, $\texttt{Terms(PT)} = T^{0000}_{x_1x_2x_3x_4} = \{\lambda x_1x_2x_3.x_1(\lambda x_4.x_1(\lambda x_4'.x_2x_4)x_4)x_3, \lambda x_1x_2x_3.x_1(\lambda x_4.x_1(\lambda x_4'.x_2x_4')x_4)x_3\}$.

The following result proves the correctness of Algorithms 1.12 and 1.13. Note that the set of long nomal inhabitants produced by Algorithm 1.13 may contain terms with bounded-variable clashes, but is of course equivalent (modulo α-conversion) to a set of long normal inhabitants without bounded-variable clashes.

Proposition 1.15

(i) If M is a long normal inhabitant of a type φ, then $\texttt{PT}(M)$ *is a valid proof-tree for φ.*

(ii) If PT *is a valid proof-tree for a type φ, then every member of* Terms(PT) *is a closed long normal inhabitant of φ.*

Furthermore, the two algorithms are complementary in the sense that for every closed long normal inhabitant M of φ there is $M \in \texttt{Terms(PT}(M))$.

Proof We begin the proof of *(i)* showing that $\texttt{PT}(M)$ can in fact be obtained applying the rules for constructing proof-trees in Definition 1.8, and showing that both conditions in Definition 1.9 are satisfied by $\texttt{PT}(M)$. For that, consider any occurrence $\overline{x}$ (not in an abstraction-sequence) of a variable x in M. Suppose that x has type $\alpha = (\alpha_{11} \to \ldots \to \alpha_{1m_1} \to A_1) \to \ldots \to (\alpha_{k1} \to \ldots \to \alpha_{km_k} \to A_k) \to A$, let p be the primitive part in $\texttt{tree}(\varphi)$ that corresponds to x, i.e. p is the primitive part in the top of $\texttt{t}(\alpha)$, and finally let $\overline{p}$ be the occurrence of p in $\texttt{PT}(M)$ created by the replacement of occurrence $\overline{x}$ of x by p during the application of algorithm PT( ).

It follows from the Formula-tree Algorithm and from the structure of α, that p has arity $k \geq 0$ with head-variable A and tail-variables $A_1, \ldots, A_k$. Furthermore, for $1 \leq j \leq k$, p has exactly m_j descendents at branch j. Thus, by Lemma 1.6, the occurrence $\overline{x}$ of x in M occurs exactly with k arguments, which for $1 \leq j \leq k$ are respectively of the form $\lambda y_{j1} \ldots y_{jm_j}.N_j$ and such that N_j is of atomic type A_j. Due to their atomic type, each N_j has to be of the form $z_jP_{j1} \ldots P_{js_j}$, $s_j \geq 0$. Thus, $\overline{p}$ will have exactly k arguments, $\overline{p}_1, \ldots, \overline{p}_k$, in $\texttt{PT}(M)$ which are respectively the primitive parts corresponding to z_1 in N_1,…and to z_k in N_k. This, satisfies the first condition in definition 1.9. Furthermore, since $N_1, \ldots, N_k$ have types $A_1, \ldots, A_k$, respectively, we infer that the primitive parts corresponding to $z_1, \ldots, z_k$, i.e. $\overline{p}_1, \ldots, \overline{p}_k$, have head-variables $A_1, \ldots, A_k$, respectively, so, by the formation rules in Definition 1.8, they fit as arguments for $\overline{p}$.

Finally, we have to show that also the second condition in Definition 1.9 is satisfied by $\texttt{PT}(M)$. Consider $\overline{p}$; we have to show that if p is a descendent at branch i of some primitive part p' in $\texttt{tree}(\varphi)$, then there is some occurrence of $\overline{p}'$ in $\texttt{PT}(M)$, such that $\overline{p}$ occurs in the subtree of $\texttt{PT}(M)$ rooted in the ith branch of $\overline{p}'$.

Suppose $p = p_0$, the primitive part in the root of $\texttt{tree}(\varphi)$, then there is no such p' and the second condition in Definition 1.9 is trivially true.

Suppose $p \neq p_0$ and let x be the variable corresponding to p. Since M is closed, $\overline{x}$ occurs in the scope of some abstraction sequence $\vec{x}$ containing x. If $\vec{x}$ is the initial abstraction sequence in M, then p

descends from $p' = p_0$ at branch 1 in $\texttt{tree}(\varphi)$ and the second condition in Definition 1.9 is satisfied, since p_0 is the primitive part in the root of $\texttt{PT}(M)$. Otherwise, $\vec{x}$ is the initial abstraction sequence of the ith argument of a subterm of M of the form $yN_1 \ldots N_k$ and $i \in \{1, \ldots, k\}$, $k \geq 1$. Then, p descends in $\texttt{tree}(\varphi)$ from the ith branch/tail-variable of the primitive part p' that corresponds to y. Since, $\overline{x}$ occurs in the ith argument of an occurrence of y, we conclude that $\overline{p}$ occurs in the subtree rooted at the ith branch of an occurrence of p' in $\texttt{PT}(M)$.

For *(ii)* we begin by noting that the only occurrence of p_0 in $\texttt{PT}$ is in its root and that p_0 has arity 1. Therefore the expression constructed in part *(a)* of Algorithm 1.13 is of the form $x_0(T)$ and that $T = \lambda x_{i_1} \ldots x_{i_k}.N$, where $p_{i_1}, \ldots, p_{i_k}$ are the descendents of p_0 (at branch 1) in $\texttt{tree}(\varphi)$ and such that N corresponds to the subtree beneath p_0 in $\texttt{PT}$.

The term $x_0(T)$ was constructed starting from an application of variables, by introduction of abstraction-sequences before subterms in argument position, but never in function position. So it is surely a λ-term in β-normal form and consequently the same is true for the term-scheme T, which is the result after step *(b)*.

Now, consider any occurrence $\overline{x}_i$ (not in an abstraction-sequence) of a variable x_i in T, which is due to an occurrence $\overline{p}_i$ in $\texttt{PT}$. Let p_m be the primitive part of arity $l \geq 1$ such that, in $\texttt{tree}(\varphi)$, p_i is a descendent of p_m at some branch j. The second condition in the definition of valid proof-trees guarantees that there is at least one occurrence $\overline{p}_m$ in $\texttt{PT}$ such that the $\overline{p}_i$ is in the subtree rooted in the jth branch of p_m. This is reflected in T as an occurrence of the variable x_m with l arguments and such that the occurrence $\overline{x}_i$ is in the jth argument of x_m, which begins with an abstraction sequence in which x_i occurs, binding thus the occurrence $\overline{x}_i$ (if not already binded[5]). We conclude that T is closed.

Finally, denote by φ_i, for $1 \leq i \leq n$, the subtype of φ such that the subtree $\texttt{t}(\varphi_i)$ in $\texttt{tree}(\varphi)$ has primitive part p_i at its top (cf. Algorithm 1.4). Now, consider any occurrence $\overline{p}_s$ of a primitive part p_s of arity $l \geq 1$ in $\texttt{PT}$, let $\texttt{PT}'$ be the subtree of $\texttt{PT}$ rooted in branch $j \in \{1, \ldots, l\}$ of $\overline{p}_s$, let A be the jth tail-variable of p_s (and consequently the head-variable of the primitive part in the top of $\texttt{PT}'$) and finally let $p_{j_1}, \ldots, p_{j_t}$, $t \geq 0$, be the descendents of p_s at branch j in $\texttt{tree}(\varphi)$. It follows from the algorithm $\texttt{Terms}$ that the jth argument of the occurrence $\overline{x}_s$ in $x_0(T)$, created due to $\overline{p}_s$, is of the form $T' = \lambda x_{j_1} \ldots x_{j_t}.P$. We will show by induction on the number of occurrences of type-variables in $\texttt{PT}'$ that, for the type-context $\Gamma = \{x_1 : \varphi_1, \ldots, x_n : \varphi_n\}$ there is $\Gamma \vdash \lambda x_{j_1} \ldots x_{j_t}.P : \varphi_{j_1} \rightarrow \ldots \rightarrow \varphi_{j_t} \rightarrow A$ in $\mathbf{TA}_\lambda$, i.e. given Γ the type $\varphi_{j_1} \rightarrow \ldots \rightarrow \varphi_{j_t} \rightarrow A$ can be assigned to $\lambda x_{j_1} \ldots x_{j_t}.P$, cf. [9, Chapter 2]. Then, taking for $\overline{p}_s$ the unique occurrence of p_0 in $\mathbf{PT}$, one has $\Gamma \vdash \lambda x_{i_1} \ldots x_{i_k}.N : \varphi_{i_1} \rightarrow \ldots \rightarrow \varphi_{i_k} \rightarrow A_0$, where A_0 is the tail-variable of p_0, i.e. $\Gamma \vdash T : \varphi$. Furthermore, since T is closed, we conclude that also $\vdash T : \varphi$ is true in $\mathbf{TA}_\lambda$. On the other hand, T is clearly long, since every variable x_i of type $\varphi^1 \rightarrow \ldots \rightarrow \varphi^l \rightarrow A$ appears always with exactly l arguments, due to the fact that the corresponding primitive part p_i has l tail-variables.

We begin the induction proof with the case $\mathbf{PT}' = p_i$, $i \in \{1, \ldots, n\}$ and $p_i = \begin{array}{c} A \\ | \end{array}$, hence $\varphi_i = A$. Since p_i has no tail-variable, there are no primitive parts descending from p_i in $\texttt{tree}(\varphi)$, thus $\lambda x_{j_1} \ldots x_{j_t}.P = \lambda x_{j_1} \ldots x_{j_t}.x_i$ and $\Gamma \vdash \lambda x_{j_1} \ldots x_{j_t}.x_i : \varphi_{j_1} \rightarrow \ldots \rightarrow \varphi_{j_t} \rightarrow A$ is true in $\mathbf{TA}_\lambda$.

Now, let the primitive part in the top of $\mathbf{PT}'$ be p_i of arity $l \geq 1$. Furthermore, for $1 \leq s \leq l$ let $p_{s_1}, \ldots, p_{s_{l_s}}$ be the descendents (in $\texttt{tree}(\varphi)$) from p_i at branch s/tail-variable A_s, let $\lambda\vec{x}_s$ be the abstraction-sequence $\lambda x_{s_1} \ldots x_{s_{l_s}}$ and let $\varphi^s = \varphi_{s_1} \rightarrow \ldots \rightarrow \varphi_{s_{l_s}} \rightarrow A_s$. Then, $\varphi_i = \varphi^1 \rightarrow \ldots \rightarrow \varphi^l \rightarrow A$ and $P = x_i(\lambda\vec{x}_1.P_1)\ldots(\lambda\vec{x}_l.P_l)$, where each $\lambda\vec{x}_s.P_s$ is constructed from subtree $\mathbf{PT}_s$ rooted in the sth tail-variable of p_i. In order to show that $\Gamma \vdash \lambda x_{j_1} \ldots x_{j_t}.P : \varphi_{j_1} \rightarrow \ldots \rightarrow \varphi_{j_t} \rightarrow A$ is true in $\mathbf{TA}_\lambda$, it suffices to show that for $1 \leq s \leq l$, given context Γ the type φ^s can be assigned to $\lambda\vec{x}_s.P_s$ in $\mathbf{TA}_\lambda$. This follows from the induction hypothesis applied to $\mathbf{PT}_s$, thus finishing the induction.

To finish the proof of part *(ii)* first note that for $n \geq 1$ and T resulting from *(b)* of $\texttt{Terms}$ Algorithm,

[5]The same variable name may appear in more than one abstraction, corresponding to different occurrences of the same primitive part.

every term $M \in T^{0\ldots0}_{x_1\ldots x_n}$ is a closed term (for this it suffices to show by structural induction on N, that for any term N with variables in $\{x_1, \ldots, x_n\}$, if $M \in N^{k_1\ldots k_n}_{x_1\ldots x_n}$, then for the set $FV(M)$ of free variables of M, one has $FV(M) \subseteq \{x^j_i \mid 1 \le i \le n, 0 \le j \le k_i - 1\} \cup \{\overline{x}_i \mid 1 \le i \le n, x_i \in FV(N), k_i = 0\}$, and consequently for $M \in T^{0\ldots0}_{x_1\ldots x_n}$ there is $FV(M) = \emptyset$). On the other hand, we know that there is a deduction of $T : \varphi$ in $\mathbf{TA}_\lambda$. This deduction can then easily be changed into a deduction of $M : \varphi$, assigning identical types to variables with the same index.

The remaining claim that for every closed long normal inhabitant M of φ there is $M \in \texttt{Terms}(\texttt{PT}(M))$, included here for the sake of completeness, is a consequence of the symmetry of the two algorithms. •

References

[1] H. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, eds., *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pp. 117–309. Oxford Science Publications, 1992.

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

[3] S. Broda and L. Damas. Counting a type's principal inhabitants. Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications, *TLCA'99*, *Lecture Notes in Computer Science* 1581, pág. 69-82, Springer Verlag, 1999.

[4] S. Broda, and L. Damas. On the structure of normal λ-terms having a certain type. *Proceedings of 7th WoLLIC'2000*, pp. 33–43, 2000. Copy in `http://www.dcc.fc.up.pt/~ sbb/wollic00.ps`.

[5] A. Church, A formulation of the simple theory of types. *Journal of Symbolic Logic*, **5**, 56–68, 1940.

[6] H. B. Curry. Functionality in combinatory logic. *Proc. Nat. Acad. Science USA*, **20**, 584–590, 1934.

[7] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, **57**, 795–807, 1992.

[8] G. Gentzen. Untersuchungen über das logische Schliessen. *Math. Zeitschrift*, **39**, 176–210, 1935.

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

[10] S. Hirokawa. Note: Infiniteness of proof(α) is polynomial-space complete. *Theoretical Computer Science*, 206:331–339, 1998.

[11] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[12] D. Prawitz. *Natural Deduction.* Almqvist and Wiksell, Sweden, 1965.

[13] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.