Pure Type Systems with de Bruijn indices [⋆]

Fairouz Kamareddine[1], and Alejandro Ríos[2]

[1] Computing and Electrical Engineering, Heriot-Watt Univ., Riccarton, Edinburgh EH14 4AS, Scotland, `fairouz@cee.hw.ac.uk`

[2] Department of Computer Science, University of Buenos Aires, Pabellón I - Ciudad Universitaria (1428) Buenos Aires, Argentina, `rios@dc.uba.ar`

Abstract. Nowadays, type theory has many applications and is used in many different disciplines. Within computer science, logic and mathematics, there are many different type systems. They serve several purposes, and are formulated in various ways. A general framework called *Pure Type Systems* (PTSs for short) has been introduced independently by Terlouw and Berardi in 1988 and 1989, in order to provide a unified formalism in which many type systems can be represented. In particular, PTSs allow the representation of the simple theory of types, the polymophic theory of types, the dependent theory of types and various other well-known type systems such as the Edinburgh Logical Frameworks LF and the Automath system.

Pure Type Systems are usually presented using variable names. In this article, we present a formulation of PTSs with de Bruijn indices. De Bruijn indices [6] avoid the problems caused by variable names during the implementation of type systems. We show that PTSs with variable names and PTSs with de Bruijn indices are isomorphic. This isomorphism enables us to answer questions about PTSs with de Bruijn indices including confluence, termination (strong normalisation) and safety (subject reduction).

1 Introduction

The explosion of new type theories and their applications in the twentieth century is fascinating. This is not surprising since type theory is considered as an important foundation for logic, the formalisation of mathematics and the design and implementation of theorem proving and programming languages. Of course, for every new type theory many questions need to be answered before that theory can be useful for some applications. Of these questions, we mention issues like termination and safety. For example, in programming terms, safety can be described as the property that specifies that if a program has a certain type and if this program is evaluated to a certain value (or another program), then this result itself has the same type. Termination can be described by the property that if a program is typable, then this program terminates (does not loop).

Some of these properties are hard to establish and it is hence desirable to generalise proofs from one system to the other if at all possible. Hence, a general framework in which different type systems can be described may turn out to be useful for providing general criteria and results concerning the various systems. A general framework also helps comparing the different systems. In 1988 and 1989, a general framework was given independently by Terlouw and Berardi in [5, 21] which classifies different known type theories. This framework is known as the *Pure Type Systems* (PTSs for short) framework. In [4], a description of PTSs and of a cube of eight different systems that are all PTSs can be found. Important type systems that are PTSs include Church's simply typed λ-calculus [7] and the calculus of constructions [8,9] which are also systems of the Barendregt cube [4].

As to how types are written within the terms, there are basically two type disciplines: the implicit and the explicit. The implicit style, also known as typing à la Curry, does not annotate variables with types. For example, the identity function is written as in the type-free case, as

[⋆] This work was supported by EPSRC grants GR/K25014, GR/L36963 and GR/L15685. We are grateful for the anonymous referees for their thorough reading and valuable comments.

$\lambda x.x$. The type of terms however is found using the typing rules of the system in use. The explicit style, also known as typing à la Church, does annotate variables and the identity function may be written as $\lambda x : \texttt{Bool}.x$ to represent identity over booleans. PTSs are based on typing à la Church and this is the discipline we consider in this paper.

So far, we know that there are two type disciplines of which we consider the explicitly typed discipline, and that there are various type systems many of which fall under the PTSs framework which we study in this paper. Besides the questions of what is the type system under consideration and what is the type discipline, there is another important question, namely: what is the variable discipline assumed by the system. There are many disciplines concerned with variables:

– The use of named variables as is usually assumed in many books of the λ-calculus and type theory. For example (ignoring types) $\lambda x.x$ represents the identity function. In the named variable discipline, substitution can be a cumbersome operation due to variable manipulation and renaming. There are some approaches used to avoid either the problem or variables themselves. We mention next some of these methods that are used in the formalisation of λ-calculus and type theory and in the implementation of programming languages and theorem provers.
– The use of combinatory logic which is equivalent to the λ-calculus but does not use variable names. For example, in combinatory logic, the identity function $\lambda x.x$ is written as I where Ia reduces to a. In fact, every term is a combinator and no variables need to be introduced. It is however less intuitive to understand what the combinators are doing especially in really large terms. Combinators looked very promising in the 1970s and 1980s when Turner developed the language Miranda and when Hughes developed the notion of super-combinators. We will not study combinators in this article. The interested reader can refer to [11].
– The use of de Bruijn indices which avoid clashes of variable names and therefore neither α-conversion nor Barendregt's convention are needed. The identity function will be written as $\lambda.1$ using de Bruijn indices. The 1 refers to the number of λs before reaching the binding λ. De Bruijn indices are explained in detail in Section 4.1. De Bruijn indices are extensively used in the implementation of programming languages and theorem provers.
– There are many other approaches to avoid the problem of named variables in type theory. For example, there is the higher order abstract syntax approach [19] which avoids formalising the renaming of variables in order to prevent unintended capture during substitution. There is also another approach due to Coquand and used in formalisations of PTSs in LEGO [16] where free variables are syntactically distinguished from bound ones and where explicit α-conversion of named variables is not necessary in the theory of reduction, conversion and typing.

There are advantages and disadvantages to each of the above mentioned approaches. This paper is not about what is the best approach to represent variables in type theory. The last two decades has seen much progress in this field with more work carried out on higher order abstract syntax, or on defending the use of named variables (and the variable convention), or on defending the need for newer disciplines of variables. Our paper is concerned with writing PTSs (which are usually written using variable names) using de Bruijn indices which seems to be the most used formulation in many implementations of type theory and theorem proving. Translations between variable names and de Bruijn indices have received attention in the past for the lambda calculus [20, 14], but never before has there been a formulation of PTSs using de Bruijn indices.

Now that we have settled the type discipline to explicit, the type systems to PTSs and the variable discipline to de Bruijn indices, we propose to write PTSs using de Bruijn indices. All the desirable properties of PTSs (such as termination and safety) have been established for PTSs with variable names and there does not yet exist a formulation of PTSs with de Bruijn indices for which the desirable properties have been established.

We believe that as far as possible, important theoretical properties like termination and safety should be established before a system can be the basis of an implementation. In this paper, we will give such a formulation of PTSs with de Bruijn indices and we show that our formulation is isomorphic to the formulation with variable names. This isomorphism will be used to establish that our formulation of PTSs with de Bruijn indices satisfies the properties mentioned above. The direct result of our work is that on one hand, one can now find a formulation of PTSs with de Bruijn indices for which desirable properties have been proved, and on the other hand, our formulation can be directly implemented because it is written with de Bruijn indices which do not lead to the problematic features that result from variable names.

This paper is divided as follows: In Section 2 we introduce the formal machinery needed. In Section 3 we recall the Pure Type Systems with variable names as presented in [4] and some of their properties. In Section 4 we introduce the Pure Type Systems with de Bruijn indices and prove some properties concerning free variables. In Section 5 we establish the isomorphisms between PTS's and their de Bruijn versions. In Section 6 we use the isomorphism to establish the properties of PTSs with de Bruijn indices.

2 Formal Machinery

We assume familiarity with the λ-calculus as in [3]. In this section, we will introduce some machinery that will be used in the rest of this paper.

Definition 1 (Reduction Notations) *Let S be a set and R a binary relation on S. We denote R by $\rightarrow_R$ and call it a reduction notion. We use the following notations and definitions:*

1. *$\twoheadrightarrow_R$ or just $\twoheadrightarrow$ is the reflexive and transitive closure of $\rightarrow_R$. When $A \twoheadrightarrow_R B$ we say there exists an R-*reduction sequence *from A to B.*
2. *$=_R$ is the reflexive, symmetric and transitive closure of $\rightarrow_R$. That is, $=_R$ is the least equivalence relation containing $\rightarrow_R$.*
3. *$\equiv$ is syntactic identity, and $A \equiv B$ means A and B are syntactically identical.*
4. *$A \in S$ is an R-*normal form *(R-nf for short) if there is no $B \in S$ such that $A \rightarrow_R B$.*
5. *We say that B* is an R-normal form of A *or A* has B as R-normal form *if B is an R-normal form and $A =_R B$.*

Expressions can be evaluated in different orders. For example, we could evaluate 2+3+4 by evaluating (2+3)+4 or 2+(3+4). We would like to get the same result either way. The following definition helps us describe this phenomenon:

Definition 2 (Confluence and Church Rosser) *Let R be a notion of reduction on S. We say that R is confluent (or Church Rosser CR) if $\twoheadrightarrow_R$ satisfies the property:*
$\forall A, B, C \in S\ \exists D \in S\ : (A \twoheadrightarrow_R B \wedge A \twoheadrightarrow_R C) \Rightarrow (B \twoheadrightarrow_R D \wedge C \twoheadrightarrow_R D)$.

Theorem 3 *Let R be a notion of reduction that is CR. The following holds:*

- *Let $A =_R B$ then*
  - *there is a C such that $A \twoheadrightarrow_R C$ and $B \twoheadrightarrow_R C$.*[1]
  - *if B is in R-normal form, then $A \twoheadrightarrow_R B$.*
  - *either A and B do not have R-normal forms or A and B have the same R-normal form.*
  - *if A and B are in R-normal forms then $A \equiv B$.*
- *If A has R-normal forms B and C, then $B \equiv C$. Hence, we speak of the R-normal form of A and denote it by $R(A)$.*

A second very important concern of reduction (or rewrite) notions is that of *termination.* We are interested in knowing if our rewriting of a particular expression will terminate or will go indefinitely. For example, the rule $n \rightarrow n+1$ applied to 1 will not terminate. Termination is a crucial property for implementation purposes. If an expression does not always terminate, perhaps it can terminate with some careful ordering of rules. Those expressions that will never terminate are disastrous for computation. The following definition introduces notions related to termination:

Definition 4 (Normalisation) *Let R be a reduction notion on S. We say that:*

[1] Sometimes, this is referred to as the confluence property. We have however identified Church Rosser and Confluence.

– *A is R* strongly normalising *if there are no infinite R-reduction sequences starting at A.*
– *R is* strongly normalising *(SN) if there is no infinite sequence $(A_i)_{i\geq 0}$ in S such that $A_i \to_R A_{i+1}$ for all $i \geq 0$. I.e. every A in S is R strongly normalising.*
– *R is* weakly normalising *(WN) if every $A \in S$ has an R-normal form.*

When no confusion can arise, R is omitted and we speak simply of normal forms or normalisation.

Strong normalisation implies weak normalisation and therefore the existence of normal forms.

3 Pure Type Systems with variable names

In this section, we will review PTSs with variable names and some of their properties.

3.1 Syntax and Rules of PTSs with variable names

Definition 5 *The* set of pseudo-terms *$\mathcal{T}$, is generated by the grammar:*
$\mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid (\mathcal{T}\,\mathcal{T}) \mid (\lambda\mathcal{V} : \mathcal{T}.\mathcal{T}) \mid (\Pi\mathcal{V} : \mathcal{T}.\mathcal{T})$, where $\mathcal{V}$ is the infinite set of variables $\{v_1, v_2, v_3, \dots\}$ and $\mathcal{C}$ a set of constants over which, $c, c_1, \dots$ range and containing two special sorts $*$ and $\Box$. We use $A, B, \dots$ to range over $\mathcal{T}$ and $x, y, z, \dots$ to range over $\mathcal{V}$.

Definition 6 (Notational convention) *We use the following notational conventions:*

1. *Throughout, we take π to represent either λ or Π.*
2. *Functional application associates to the left. So ABC denotes $((AB)C)$.*
3. *The body or scope of a π is anything that comes after it. So, instead of $(\pi v : A.(A_1 A_2 \dots A_n))$, we write $\pi v : A.A_1 A_2 \dots A_n$.*
4. *A sequence of π's is compressed to one, so for example, $\lambda x : A\, y : B\, z : C.t$ denotes $\lambda x : A.(\lambda y : B.(\lambda z : C.t))$.*

Therefore, application has priority over abstraction: $\pi x : A.yz$ means $\pi x : A.(yz)$ and not $(\pi x : A.y)z$.
π is a variable binder, just like $\forall$ in logic. Hence we define *free* and *bound* variables.

Definition 7 (Free and Bound variables) *For a term C, the set of free variables $FV(C)$, and the set of bound variables $BV(C)$, are defined inductively as follows:*

$$\begin{array}{llll}
FV(v) & =_{def} \{v\} & BV(v) & =_{def} \emptyset \quad \emptyset \textit{ is the empty set} \\
FV(c) & =_{def} \emptyset & BV(c) & =_{def} \emptyset \\
FV(\pi v : A.B) & =_{def} (FV(B) \setminus \{v\}) \cup FV(A) & BV(\pi v : A.B) & =_{def} BV(A) \cup BV(B) \cup \{v\} \\
FV(AB) & =_{def} FV(A) \cup FV(B) & BV(AB) & =_{def} BV(A) \cup BV(B)
\end{array}$$

An occurrence of a variable v in a term is free if it is not within the scope of a $\pi v : A.$, otherwise it is bound. A *closed term* is a term in which all variables are bound. We write $A[x := B]$ to denote the term where all the free occurrences of x in A have been replaced by B.
An important notion for rewriting relations is that of *compatibility*:

Definition 8 (Compatibility for PTSs) *We say that a binary relation R on a PTS is compatible iff for all terms A, B, and variable v, the following holds:*

$$\frac{(A,B) \in R}{(AC, BC) \in R}(a_1) \qquad \frac{(A,B) \in R}{(CA, CB) \in R}(a_2)$$

$$\frac{(A,B) \in R}{(\pi v : C.A, \pi v : C.B) \in R}(b_1) \qquad \frac{(A,B) \in R}{(\pi v : A.C, \pi v : B.C) \in R}(b_2)$$

α-reduction identifies terms up to variable renaming:

Definition 9 (Alpha reduction) α-reduction $\to_\alpha$ *is defined to be the least compatible relation generated by the axiom:*

$$(\alpha) \qquad \pi v : A.B \to_\alpha \pi v' : A.B[v := v'] \qquad \textit{where } v' \notin FV(B)$$

Now one can follow Definition 1 to define $=_\alpha$ (α-equivalences), $\twoheadrightarrow_\alpha$, etc.

As usually done with calculi with named variables we will identify α-equivalent terms and we will not use a special notation to differentiate terms and classes of terms. However, when we want to stress the fact that two terms, say A and B are α equivalent but may not be identical we will eventually write $A =_\alpha B$ instead of $A = B$. Furthermore, we assume the Barendregt variable convention which is formally stated as follows:

Convention 10 (VC: Barendregt's Convention) *Names of bound variables will always be chosen such that they differ from the free ones in a term. Moreover, different π's have different variables as subscript. Hence, we will not have $(\pi x : A.x)x$, but $(\pi y : A.y)x$ instead.*

Lemma 11 (Substitution for variable names) *Let $A, B, C \in \mathcal{T}$, $x, y \in \mathcal{V}$. For $x \neq y$ and $x \notin FV(C)$, we have that: $A[x := B][y := C] = A[y := C][x := B[y := C]]$.*

Definition 12 (Beta reduction) β-reduction $\to_\beta$ *is the least compatible relation on $\mathcal{T}$ generated by*

$$(\beta) \qquad (\lambda x : A.B)C \to B[x := C]$$

Here is a lemma about the interaction of β-reduction and substitution. Note that in the first case, $\to_\beta$ is mapped into $\twoheadrightarrow_\beta$. The reason being that x may occur n times (for $n \geq 0$) as a free variable in A and hence the reduction will be repeated following the number of occurrences of x.

Lemma 13 *Let $A, B, C, D \in \mathcal{T}$.*

1. *If $C \to_\beta D$ then $A[x := C] \twoheadrightarrow_\beta A[x := D]$.*
2. *If $A \to_\beta B$ then $A[x := C] \to_\beta B[x := C]$.*

Proof: By induction on the structure of A for 1, on the generation of $A \to_\beta B$ for 2. ⊠

Now, we define some machinery needed for typing:

Definition 14

1. *A* statement *is of the form $A : B$ with $A, B \in \mathcal{T}$. We call A the* subject *and B the* predicate *of $A : B$.*
2. *A* declaration *is of the form $x : A$ with $A \in \mathcal{T}$ and $x \in \mathcal{V}$.*
3. *A* pseudo-context *is a finite ordered sequence of declarations, all with distinct subjects. We use $\Gamma, \Delta, \Gamma', \Gamma_1, \Gamma_2, \ldots$ to range over pseudo-contexts. The* empty *context is denoted by either $<>$ or nothing at all if no confusion can arise.*
4. *If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then $\Gamma, x : B = x_1 : A_1, \ldots, x_n : A_n, x : B$ and $dom(\Gamma) = \{x_1, \ldots, x_n\}$.*

Definition 15 *A* type assignment *relation is a relation between pseudo-contexts and pairs of pseudo-terms written as $\Gamma \vdash A : B$. The* rules of type assignment *establish which* judgments *$\Gamma \vdash A : B$ can be derived. A judgement $\Gamma \vdash A : B$ states that $A : B$ can be derived from the pseudo-context Γ.*

Definition 16 *Let Γ be a pseudo-context and $\vdash$ be a type assignment relation.*

1. *Γ is called legal if $\exists A, B \in \mathcal{T}$ such that $\Gamma \vdash A : B$.*
2. *$A \in \mathcal{T}$ is called a Γ-`term` if $\exists B \in \mathcal{T}$ such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.*
   We take Γ-terms $= \{A \in \mathcal{T}$ such that $\exists B \in \mathcal{T}$ and $\Gamma \vdash A : B \vee \Gamma \vdash B : A\}$.
3. *$A \in \mathcal{T}$ is called legal if $\exists \Gamma$ such that $A \in \Gamma$-terms.*

The next definition will introduce a generalised family of type systems called Pure Type Systems (or PTSs). A PTS can be distinguished from another by:

- The set of sorts $\mathcal{S}$ which is a subset of the constants $\mathcal{C}$. The two most used sorts are $*$ and $\Box$. $A : *$ can be read as A is a type. $A : \Box$ can be read as A is a kind.
- The set of axioms $\mathcal{A}$ of the form $c : s$ which type special constants. $* : \Box$ is an example of such an axiom.
- A set of rules $\mathcal{R}$ which restrict type formation as to allow/disallow notions like polymorphim, dependent types, etc.

Definition 17 *The* specification *of a PTS is a triple $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a subset of $\mathcal{C}$, called the* sorts. *$\mathcal{A}$ is a set of* axioms *of the form $c : s$ with $c \in \mathcal{C}$ and $s \in \mathcal{S}$ and $\mathcal{R}$ is a set of* rules *of the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$.*

Definition 18 *The notion of type derivation, denoted $\Gamma \vdash_{\lambda S} A : B$ (or simply $\Gamma \vdash A : B$), in a PTS whose specification is $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, is axiomatised by the axioms and rules of Figure 1.*

$$
\begin{array}{lll}
\text{(axiom)} & <>\vdash c : s & \text{if } \; c : s \in \mathcal{A} \\[2ex]
\text{(start)} & \dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} & \text{if } \; x \notin dom(\Gamma) \\[2ex]
\text{(weakening)} & \dfrac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C} & \text{if } \; x \notin dom(\Gamma) \\[2ex]
\text{(product)} & \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_3} & \text{if } \; (s_1, s_2, s_3) \in \mathcal{R} \\[2ex]
\text{(application)} & \dfrac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash C : A}{\Gamma \vdash F\, C : B[x := C]} & \\[2ex]
\text{(abstraction)} & \dfrac{\Gamma, x : A \vdash C : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.C) : (\Pi x : A.B)} & \\[2ex]
\text{(conversion)} & \dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'} &
\end{array}
$$

Fig. 1. PTSs with variables names

Each of the eight systems of the cube is obtained by taking $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{* : \Box\}$, and $\mathcal{R}$ to be a set of rules of the form (s_1, s_2, s_2) for $s_1, s_2 \in \{*, \Box\}$. This means that for the cube, the only possible (s_1, s_2, s_2) rules in the set R are elements of the set: $\{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$. The basic system is the one where $(s_1, s_2, s_2) = (*, *, *)$ is the only possible choice. All other systems have this version of the formation rules, plus one or more other combinations of $(*, \Box, \Box)$, $(\Box, *, *)$ and $(\Box, \Box, \Box)$ for (s_1, s_2, s_2). See Figures 2 and 3. See also Page 192 of [4].

Note that as in the cube there are only two sorts, $*$ and $\Box$, and as each set $\mathcal{R}$ must contain $(*, *, *)$, there are only eight possible different systems for the Cube. An important aspect of the Cube is that it provides a factorisation of the expressive power of the Calculus of Constructions into three features: *polymorphism, type constructors*, and *dependent types*:

- $(*, *, *)$ is the basic rule that forms types. All type systems of the Cube have this rule.

$\lambda_{\rightarrow}$	$(*,*,*)$			
$\lambda 2$	$(*,*,*)$	$(\Box,*,*)$		
λP	$(*,*,*)$		$(*,\Box,\Box)$	
$\lambda P2$	$(*,*,*)$	$(\Box,*,*)$	$(*,\Box,\Box)$	
$\lambda\underline{\omega}$	$(*,*,*)$			$(\Box,\Box,\Box)$
$\lambda\omega$	$(*,*,*)$	$(\Box,*,*)$		$(\Box,\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*,*)$		$(*,\Box,\Box)$	$(\Box,\Box,\Box)$
$\lambda P\omega = \lambda C$	$(*,*,*)$	$(\Box,*,*)$	$(*,\Box,\Box)$	$(\Box,\Box,\Box)$

Fig. 2. Different type formation condition

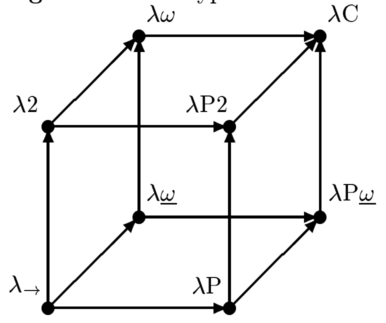

Fig. 3. The cube

- $(\Box,*,*)$ is the rule that takes care of polymorphism. Girard's System (also known as $\lambda 2$) is the weakest system on the Cube that features this rule.
- $(\Box,\Box,\Box)$ takes care of type constructors. The system $\lambda\underline{\omega}$ is the weakest system on the Cube that features this rule.
- $(*,\Box,\Box)$ takes care of term dependent types. The system λP is the weakest system on the Cube that features this rule.

Many other well-known type systems, like AUTOMATH [18], LF [10], and ML [17] can be more or less related to one of the systems of the Barendregt Cube.

3.2 Properties of PTSs with variable names

Now, we list some of the properties of PTSs with variable names (see [4] for proofs). In Section 6, we will establish these properties for PTSs with de Bruijn indices.

Lemma 19 *Let $A, B \in \mathcal{T}$. If $A \rightarrow_\beta B$ then $FV(B) \subseteq FV(A)$.*

Theorem 20 (The Church Rosser Theorem for PTSs with variable names) *If $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$ then there exists D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.*

Lemma 21 (Free variable lemma) *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ such that $\Gamma \vdash B : C$. The following hold (proof is by induction on the derivation $\Gamma \vdash B : C$):*

1. *$FV(B), FV(C) \subseteq \{x_1, \ldots, x_n\}$.*
2. *$FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $1 \leq i \leq n$.*

Theorem 22 (Subject Reduction (SR) for PTSs with variable names) *If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$.*

The next definition introduces the notion of singly sorted PTSs, which impose that the special constants have unique sorts as types and which imply the unicity of types.

Definition 23 *Let $\lambda S = \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a given PTS. λS is called singly sorted if:*

1. *$(c : s_1), (c : s_2) \in \mathcal{A}$ implies $s_1 = s_2$.*
2. *$(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R}$ implies $s_3 = s_3'$.*

Lemma 24 (Unicity of types for singly sorted PTSs with variable names) *In any singly sorted PTS, the following holds:*

1. *If $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$ then $B_1 =_\beta B_2$.*
2. *If $\Gamma \vdash A : B$ and $\Gamma \vdash A' : B'$ then $A =_\beta A'$ implies $B =_\beta B'$.*
3. *If $\Gamma \vdash B : s$, $B =_\beta B'$ and $\Gamma \vdash A' : B'$, then $\Gamma \vdash B' : s$.*

Theorem 25 (Strong normalisation for the systems of the cube) *Every legal term of the cube with variable names is strongly normalising.*

4 Pure Type Systems with de Bruijn indices

4.1 Syntax

De Bruijn noted that due to the fact that terms like $\lambda x : z.x$ and $\lambda y : z.y$ are the "same", one can find a λ-notation modulo α-conversion. That is, following de Bruijn, one can abandon variables and use indices instead. The idea of de Bruijn indices is to remove all the variables of the λ's and to replace their occurrences in the body of the term by the number which represents how many λ's one has to cross before one reaches the λ binding the particular occurrence at hand.

In the presence of free variables, a *free variable list* which orders the variables must be assumed. For example, assume we take $x, y, z, u, v, \ldots$ to be the free variable list where x comes before y which is before z, etc. Then, in order to write terms using de Bruijn indices, we use the same procedure above for all the bound variables. For a free variable however, say z, we count as far as possible the λ's in whose scope z is, and then we continue counting in the free variable list using the order assumed. The following examplifies this situation:

Example 26

1. *$\lambda x : z.x$ is replaced by $\lambda 3.1$. That is, x is removed, and the x of the body x is replaced by 1 to indicate the λ it refers to.*
2. *$\lambda x : y.xz$ and $(\lambda x : z.xz)y$ translate respectively into $\lambda 2.14$ and $(\lambda 3.14)2$.*

Now we are ready to define PTSs with de Bruijn indices.

Definition 27 *We define $\mathcal{T}$, the* set of pseudo-terms with de Bruijn indices*, by the syntax:*
$\mathcal{T} ::= \mathbb{N} \mid \mathcal{C} \mid (\mathcal{T}\,\mathcal{T}) \mid (\lambda\mathcal{T}.\mathcal{T}) \mid (\Pi\mathcal{T}.\mathcal{T})$, where $\mathcal{C}$ is a set of constants over which $c, c_1, \ldots$ range. We use $A, B, \ldots$ to range over $\mathcal{T}$ and $m, n, \ldots$ to range over $\mathbb{N}$ (positive natural numbers).

We assume conventions 1, 2, and an analogous to 3 of Definition 6 and their consequences.

The definition of compatibility (Definition 8) is changed for de Bruijn indices by replacing (b_1) and (b_2) by the following:

$$\frac{(A, B) \in R}{(\pi A.C, \pi B.C) \in R}(b_1') \qquad\qquad \frac{(A, B) \in R}{(\pi C.A, \pi C.B) \in R}(b_2')$$

4.2 Updating, Substitution and Reduction

In order to define β-reduction, we must define the substitution of a variable by a term B in a term A. Therefore, we must identify amongst the numbers of a term A those that correspond to the variable that is being substituted for and we need to update the term to be substituted in order to preserve the correct bindings of its variables.

Example 28 *Translating* $(\lambda x : v.\lambda y : v.zxy)(\lambda x : v.yx) \rightarrow_\beta \lambda u : v.z(\lambda x : v.yx)u$ *into de Bruijn indices, one gets* $(\lambda 5.\lambda 6.521)(\lambda 5.31) \rightarrow_\beta \lambda 5.4(\lambda 6.41)1$. *But, how can we carry the* β*-reduction without translating the result from variable names? The body of* $\lambda 5.\lambda 6.521$ *is* $\lambda 6.521$ *and the variable bound by the first* λ *of* $\lambda 5.\lambda 6.521$ *is the* 2. *Hence, we need to replace in* $\lambda 6.521$ *the* 2 *by* $\lambda 5.31$. *But if we simply replace* 2 *in* $\lambda 6.521$ *by* $\lambda 5.31$ *we get* $\lambda 6.5(\lambda 5.31)1$, *which is not correct. We needed to decrease* 5 *as one* λ *disappeared and to increment the free variables of* $\lambda 5.31$ *as they occur within the scope of one more* λ. *Doing all this will lead to the final result* $\lambda 5.4(\lambda 6.41)1$.

In order to define β-reduction $(\lambda C.A)B \rightarrow_\beta ?$ using de Bruijn indices. We must:

(a) find in A the occurrences $n_1, \dots n_k$ of the variable bound by the λ of $\lambda C.A$.
(b) decrease the free variables of A to reflect the disappearance of the λ from λA.
(c) replace the occurrences $n_1, \dots n_k$ in A by updated versions of B which take into account that free variables in B may appear within the scope of extra λs in A.

It will take some work to do this. Let us, in order to simplify things say that the β-rule is $(\lambda CA)B \rightarrow_\beta A\{\!\{1 \leftarrow B\}\!\}$ and let us define $A\{\!\{1 \leftarrow B\}\!\}$ in a way that all the work of $(a) - (c)$ above is carried out. We need counters described informally as follows:

1. We start traversing A (here $\lambda 6.521$) with a unique counter initialised at 1.
2. When arriving at an application node, we create a copy of the counter in order to have one counter for each branch.
3. When arriving at an abstraction node, we increment the counter.
4. When arriving at a leaf (i.e. a number):
   (a) If it is superior to the counter, we decrease the number by 1, because there will be one λ less between this number and the λ that binds it.
   (b) If the number is equal to the counter, say n, it must be replaced by B which will be found now under $n - 1$ λ's. We must therefore adjust the numbers of B so that we can modify the binding relations inside B. For this we use a family of functions that we call *updating functions.*
   (c) If the number is inferior to the value of the counter, then it is bound by a λ which is inside A, and hence the number must not be modified.

Let us define the updating functions.

Definition 29 *The* updating functions $U^i_k : T \rightarrow T$ *for* $k \geq 0$ *and* $i \geq 1$ *are defined inductively as follows:*

$$\begin{array}{ll} U^i_k(c) = c \quad \textit{for } c \in \mathcal{C} & \\ U^i_k(\pi A.B) = \pi U^i_k(A).(U^i_{k+1}(B)) & U^i_k(\mathtt{n}) = \begin{cases} \mathtt{n+i-1} & \textit{if } n > k \\ \mathtt{n} & \textit{if } n \leq k . \end{cases} \\ U^i_k(AB) = U^i_k(A)\, U^i_k(B) & \end{array}$$

The intuition behind U^i_k is the following: k tests for free variables and $i - 1$ is the value by which a variable, if free, must be incremented.

Now we define the family of substitution functions:

Definition 30 *The* substitutions at level i, *for* $i \geq 1$, *of a term* $B \in T$ *in a term* $A \in T$, *denoted* $A\{\!\{\mathtt{i} \leftarrow B\}\!\}$, *are defined inductively on* A *as follows:*

$$\begin{array}{lll} c\{\!\{\mathtt{i} \leftarrow B\}\!\} & = c \quad \textit{for } c \in \mathcal{C} & \\ (A_1A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} & = (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) & \mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} = \begin{cases} \mathtt{n-1} & \textit{if } n > i \\ U^i_0(B) & \textit{if } n = i \\ \mathtt{n} & \textit{if } n < i . \end{cases} \\ (\pi A.C)\{\!\{\mathtt{i} \leftarrow B\}\!\} & = \pi A\{\!\{\mathtt{i} \leftarrow B\}\!\}.(C\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) & \end{array}$$

The second and third equalities propagate the substitution through applications and abstractions and the first one carries out the substitution of the intended variable (when $n = i$) by the updated term. If the variable is not the intended one it must be decreased by 1 if it is free (case $n > i$) because one λ has disappeared, whereas if it is bound (case $n < i$) it must remain unaltered.

The next lemma establishes the properties of the substitutions and updating functions. The proof of this lemma is obtained by induction on A and similar to the type-free case in [13].

Lemma 31

1. *[Substitution lemma] For* $1 \le i \le n$ *we have:*
   $A\{\!\{\mathtt{i} \leftarrow B\}\!\}\{\!\{\mathtt{n} \leftarrow C\}\!\} = A\{\!\{\mathtt{n}+1 \leftarrow C\}\!\}\{\!\{\mathtt{i} \leftarrow B\{\!\{\mathtt{n}-\mathtt{i}+1 \leftarrow C\}\!\}\}\!\}$.
2. *[Distribution lemma] For* $n \le k+1$ *we have:* $U^i_k(A\{\!\{\mathtt{n} \leftarrow B\}\!\}) = U^i_{k+1}(A)\{\!\{\mathtt{n} \leftarrow U^i_{k-n+1}(B)\}\!\}$.

Case 1 is the version of Lemma 11 using de Bruijn indices.

Definition 32 (Beta reduction) β-reduction *is the least compatible reduction on* T *generated by:*

$$(\beta) \qquad (\lambda A.C)B \to_\beta C\{\!\{\mathtt{1} \leftarrow B\}\!\}$$

Remark that we use $\to_\beta$ to denote both, β-reduction on $\mathcal{T}$ and β-reduction on T. The context will always be clear enough to determine the intended reduction.

Finally, here is the version of Lemma 13 for de Bruijn indices. Note that we need not only to ensure the good passage of the β-rule through the substitutions but also through the U^i_k.

Lemma 33 *Let* A, B, C, $D \in T$.

1. *If* $C \to_\beta D$ *then* $i)$ $U^i_k(C) \to_\beta U^i_k(D)$ *and* $ii)$ $A\{\!\{\mathtt{i} \leftarrow C\}\!\} \twoheadrightarrow_\beta A\{\!\{\mathtt{i} \leftarrow D\}\!\}$.
2. *If* $A \to_\beta B$ *then* $A\{\!\{\mathtt{i} \leftarrow C\}\!\} \to_\beta B\{\!\{\mathtt{i} \leftarrow C\}\!\}$.

Proof: 1. Case $i)$ is by induction on C using Lemma 31.2. Case $ii)$ is by induction on A using $i)$. 2. Is by induction on A using Lemma 31.1. ⊠

We now define the set of free variables of a term with de Bruijn indices. We need first to define the following operations on sets of natural numbers.

Definition 34 *Let* $N \subset I\!N$ *and* $k \ge 0$. *We define:*

1. $N \setminus k = \{n-k : n \in N, n > k\}$ 2. $N + k = \{n+k : n \in N\}$
3. $N_{>k} = \{n \in N : n > k\}$ 4. $N_{<k} = \{n \in N : n < k\}$

The following properties of the above operations will be needed later and their proofs are easy.

Remark 35 *Let* N, $M \subset I\!N$ *and* k, $k' \ge 0$. *We have*

1. $(N \cup M) \setminus k = (N \setminus k) \cup (M \setminus k)$ 2. $(N \setminus k) \setminus k' = N \setminus (k + k')$
3. $(N + k) \setminus 1 = N + (k-1)$ *if* $k \ge 1$ 4. $(N \setminus 1)_{<k} = (N_{<k+1}) \setminus 1$
5. $(N \setminus 1)_{>k} = (N_{>k+1}) \setminus 1$

In the definition of free variables we give now, only the difference $\setminus$ is needed. The other operations will be useful later.

Definition 36 *The* set of free variables *of a term with de Bruijn indices is defined by induction as follows:*

$$FV(c) =_{def} \emptyset \ \text{ for } c \in \mathcal{C} \qquad FV(A\,B) =_{def} FV(A) \cup FV(B)$$
$$FV(\mathtt{n}) =_{def} \{n\} \qquad FV(\pi A.C) =_{def} FV(A) \cup (FV(C) \setminus 1)$$

Lemma 37 *For* $A \in T$ *we have* $FV(U^i_k(A)) \setminus k = (FV(A) \setminus k) + (i-1)$.

Proof: Induction on A. Use Remark 35.1 for the case $A = B\,C$ and Remark 35.2 for the case $A = \lambda B.C$. ⊠

Lemma 38 *For* A, $B \in T$ *and* $j \ge 1$, *the following hold:*

1. $FV(A\{\!\{\mathtt{j} \leftarrow B\}\!\}) = (FV(A))_{<j} \cup ((FV(A))_{>j} \setminus 1)$ *if* $j \notin FV(A)$.
2. $FV(A\{\!\{\mathtt{j} \leftarrow B\}\!\}) = (FV(A))_{<j} \cup ((FV(A))_{>j} \setminus 1) \cup (FV(B) + (j-1))$ *if* $j \in FV(A)$.

Proof: By simultaneous induction on A. Use the previous lemma for the case $A = \mathtt{j}$ and Remark 35.3, 4, 5 for the case $A = \lambda B.C$. ⊠

The following lemma on T corresponds to Lemma 19 on $\mathcal{T}$.

Lemma 39 *Let* $A, B \in T$. *If* $A \to_\beta B$ *then* $FV(B) \subseteq FV(A)$.

Corollary 40 *Let* $A, B \in T$. *If* $A \twoheadrightarrow_\beta B$ *then* $FV(B) \subseteq FV(A)$.

4.3 Rules of PTSs with de Bruijn indices

Definition 14 for PTSs with variable names changes when de Bruijn indices are used as follows:

A *(de Bruijn) pseudo-context* Γ becomes a finite ordered sequence of de Bruijn terms. We write it simply as $\Gamma = A_1, \ldots, A_n$. Statements, subject and predicate remain unchanged, and declarations disappear.

Definitions 15, 16 and 17 are the same for de Bruijn indices (except that $\mathcal{T}$ changes to T). Now, we can give the definition of PTSs using de Bruijn indices:

Definition 41 *The notion of type derivation, denoted $\Gamma \vdash_{\lambda S} A : B$ (or simply $\Gamma \vdash A : B$), in a PTS whose specification is $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, is axiomatised by the axioms and rules of Figure 4.*

$$\text{(axiom)} \quad <>\vdash c : s \qquad \text{if} \quad c : s \in \mathcal{A}$$

$$\text{(start)} \quad \frac{\Gamma \vdash A : s}{A, \Gamma \vdash \mathtt{1} : U_0^2(A)}$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash B : C \qquad \Gamma \vdash A : s}{A, \Gamma \vdash U_0^2(B) : U_0^2(C)}$$

$$\text{(product)} \quad \frac{\Gamma \vdash A : s_1 \qquad A, \Gamma \vdash B : s_2}{\Gamma \vdash (\Pi A.B) : s_3} \qquad \text{if} \quad (s_1, s_2, s_3) \in \mathcal{R}$$

$$\text{(application)} \quad \frac{\Gamma \vdash F : (\Pi A.B) \qquad \Gamma \vdash C : A}{\Gamma \vdash F\,C : B\{\!\{\mathtt{1} \leftarrow C\}\!\}}$$

$$\text{(abstraction)} \quad \frac{A, \Gamma \vdash C : B \qquad \Gamma \vdash (\Pi A.B) : s}{\Gamma \vdash (\lambda A.C) : (\Pi A.B)}$$

$$\text{(conversion)} \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}$$

Fig. 4. PTSs with de Bruijn indices

Remark that in the rules *(start), (weakening), (product), (abstraction)* the position of A with respect to Γ is reversed with respect to its position in the corresponding rules of the classical setting. However, we have chosen this presentation following the original work of type systems in de Bruijn notation (cf. [1]).

Remark also the role played by the updating U_0^2 in the rules *(start), (weakening)*. This function increases with 1 the de Bruijn indices which correspond to free variables and its occurrence in these two rules is reasonable since the corresponding contexts have been augmented by the addition of a new component.

Example 42

1. $\vdash * : \Box$	*by axiom*
2. $* \vdash 1 : *$	*by 1 and start*
3. $1, * \vdash 1 : 2$	*by 2 and start*
4. $1, 1, * \vdash 2 = U_0^2(1) : 3 = U_0^2(2)$	*by 3 twice and weakening*

The following lemma is the equivalent for de Bruijn indices, of Lemma 21.

Lemma 43 *Let $A_1, \ldots, A_n \vdash B : C$ then*
1. $FV(B), FV(C) \subseteq \{1, \ldots, n\}$ 2. *for* $0 \leq i \leq n-1$, $FV(A_{n-i}) \subseteq \{1, \ldots, i\}$.

Proof: Both items are proved by induction on the derivation.
1. For *(start)* and *(weakening)* use lemma 37. For *(application)* use lemma 38. The other rules are immediate.
2. For *(start)* and *(weakening)* use 1. The other rules are immediate. ⊠

5 The isomorphism

In the rest of this paper, we present the isomorphism between PTSs written using variable names and PTSs written using de Bruijn indices. The method is as follows:

1. We translate each term A and each context Γ written using variable names, into a term $t_{[\ldots]}(A)$ and a context $t(\Gamma)$ written with de Bruijn indices. We then prove that these translations preserve β-reduction (if in $\mathcal{T}$, $A \to_\beta B$ then in T, $t_{[\ldots]}(A) \to_\beta t_{[\ldots]}(B)$) and type assignment (if in $\mathcal{T}$, $\Gamma \vdash A : B$ then in T, $t(\Gamma) \vdash t_{[\ldots]}(A) : t_{[\ldots]}(B)$).
2. We define translations $u_{[\ldots]}$ and u in the other direction and also prove preservation of β-reduction and type assignment.
3. We prove that these translations are inverses of each other.

Notation 44

1. *We write $[x_1, \ldots, x_n]$ for the ordered list of $x_1, \ldots, x_n$.*
2. *For $i \geq 0$, we write $\overline{x}^i$ for $x_1, \ldots, x_i$ and $\underline{x}_i$ for $x_i, \ldots, x_1$.*

5.1 Translating $\mathcal{T}$ to T

Definition 45 (The translation t) *For every term $A \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1, \ldots, x_n\}$ we define $t_{[x_1,\ldots,x_n]}(A)$ by induction on A as follows:*

$$\begin{aligned}
t_{[x_1,\ldots,x_n]}(c) &=_{def} c \quad \textit{for } c \in \mathcal{C} \\
t_{[x_1,\ldots,x_n]}(v_i) &=_{def} \min\{j \textit{ such that } v_i = x_j\} \\
&\qquad \textit{Note: } \min\{j \textit{ such that } v_i = x_j\} \textit{ is interpreted as a de Bruijn index.} \\
t_{[x_1,\ldots,x_n]}(A\,B) &=_{def} t_{[x_1,\ldots,x_n]}(A) t_{[x_1,\ldots,x_n]}(B) \\
t_{[x_1,\ldots,x_n]}(\pi x : B.A) &=_{def} \pi t_{[x_1,\ldots,x_n]}(B).t_{[x,x_1,\ldots,x_n]}(A)
\end{aligned}$$

Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ be a legal context. We define:
$t(\Gamma) =_{def} t_{[x_{n-1},\ldots,x_1]}(A_n), t_{[x_{n-2},\ldots,x_1]}(A_{n-1}), \ldots, t_{[x_1]}(A_2), t_{[\,]}(A_1)$.

Remark that Definition 45 is a good definition thanks to Lemma 21.

Lemma 46 *Let $A \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1, \ldots, x_n\}$. Then $FV(t_{[x_1,\ldots,x_n]}(A)) \subseteq \{1, \ldots, n\}$.*

Proof: By induction on A. ⊠

We need to establish some lemmas before proving the preservation of type assignment. These lemmas state how the translation behaves with the updating functions, the substitutions, the β-contractions and β-equivalence.

Lemma 47 *Let $A \in \mathcal{T}$, $k \geq 0$, $i \geq 1$ and $n \geq k+i$ such that $x_{k+1}, \ldots, x_{k+i-1} \notin FV(A)$. Then $t_{[x_1,\ldots,x_n]}(A) = U_k^i(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(A))$.*

Proof: By induction on A. The case $A = c \in C$ is immediate, the case $A = B\,C$ just need the IH, the cases $A = \lambda x : B.C$ and $A = \Pi x : B.C$ are similar. Therefore, we just study:

$A = v_m$ **:** Let $j = \min\{i : v_m = x_i\}$. Then $t_{[x_1,\ldots,x_n]}(v_m) = j$.
If $j \le k$ we have $t_{[x_1,\ldots,x_n]}(A) = j = U^i_k(j) = U^i_k(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(A))$.
If $j \ge k+i$ we have $t_{[x_1,\ldots,x_n]}(A) = j = U^i_k(j-i+1) = U^i_k(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(A))$.

$A = \lambda x : B.C$ **:** We have $t_{[x_1,\ldots,x_n]}(A) = \lambda t_{[x_1,\ldots,x_n]}(B).t_{[x,x_1,\ldots,x_n]}(C) \overset{IH}{=}$
$\lambda U^i_k(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(B)).U^i_{k+1}(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(C)) =$
$U^i_k(\lambda t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(B).t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(C)) = U^i_k(t_{[x_1,\ldots,x_k,x_{k+i},\ldots,x_n]}(A))$ ⊠

Lemma 48 *Let A, $B \in \mathcal{T}$ such that the bound variables of B are not free in A and take $\overline{y}^{i-1}$ and $\overline{x}^n$ for $i \ge 1$ and $n \ge 0$. Let x be a variable not bound in B and distinct from $y_1,\ldots,y_{i-1}$ and assume $y_1,\ldots,y_{i-1} \notin FV(A)$. Then $t_{[\overline{y}^{i-1},\overline{x}^n]}(B[x := A]) = (t_{[\overline{y}^{i-1},x,\overline{x}^n]}(B))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}$.*

Proof: By induction on B. We just study the interesting cases:

$B = z \in V$ **:** We consider three cases:
If $z = x$, then $t_{[\overline{y}^{i-1},\overline{x}^n]}(B[x := A]) = t_{[\overline{y}^{i-1},\overline{x}^n]}(A) \overset{L\,47}{=}$
$U^i_0(t_{[\overline{x}^n]}(A)) = (t_{[\overline{y}^{i-1},x,\overline{x}^n]}(B))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}$
If $\{j : z = y_j\} \neq \emptyset$, let $k = \min\{j : z = y_j\}$. Then
$t_{[\overline{y}^{i-1},\overline{x}^n]}(B[x := A]) = \mathtt{k} = (t_{[\overline{y}^{i-1},x,\overline{x}^n]}(B))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}$
If $\{j : z = x_j\} \neq \emptyset$, let $k = \min\{j : z = x_j\}$. We can assume $x_k \neq x$ since the case $z = x$ has already been considered. We have
$t_{[\overline{y}^{i-1},\overline{x}^n]}(B[x := A]) = \mathtt{k} + \mathtt{i} - \mathtt{1} = \mathtt{k} + \mathtt{i}\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\} = (t_{[\overline{y}^{i-1},x,\overline{x}^n]}(B))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}$

$B = \lambda z : D.E$ **:** Remark that, since x is not bound in B, $x \neq z$. We have
$t_{[\overline{y}^{i-1},\overline{x}^n]}(B[x := A]) = \lambda t_{[\overline{y}^{i-1},\overline{x}^n]}(D[x := A]).t_{[z,\overline{y}^{i-1},\overline{x}^n]}(E[x := A]) \overset{IH}{=}$
$\lambda(t_{[\overline{y}^{i-1},x,\overline{x}^n]}(D))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}.(t_{[z,\overline{y}^{i-1},x,\overline{x}^n]}(E))\{\!\{\mathtt{i}+\mathtt{1} \leftarrow t_{[\overline{x}^n]}(A)\}\!\} =$
$(\lambda t_{[\overline{y}^{i-1},x,\overline{x}^n]}(D).t_{[z,\overline{y}^{i-1},x,\overline{x}^n]}(E))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\} = (t_{[\overline{y}^{i-1},x,\overline{x}^n]}(B))\{\!\{\mathtt{i} \leftarrow t_{[\overline{x}^n]}(A)\}\!\}$

Remark that we were able to apply the IH, because $z \notin FV(A)$, since we assumed that the bound variables of B are not free in A. ⊠

Lemma 49 *Let $A, B \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1,\ldots x_n\}$ and $A \to_\beta B$.
Then $t_{[x_1,\ldots,x_n]}(A) \to_\beta t_{[x_1,\ldots,x_n]}(B)$.*

Proof: Remark that Lemma 19 guarantees the good definition of $t_{[x_1,\ldots,x_n]}(B)$.
The proof is by induction on A. The interesting case is when A is an application and the reduction takes place at the root.
Therefore, let $A = (\lambda x : D.C)E$ and $B = C[x := E]$. We have
$t_{[x_1,\ldots,x_n]}(A) = (\lambda t_{[x_1,\ldots,x_n]}(D).t_{[x,x_1,\ldots,x_n]}(C))t_{[x_1,\ldots,x_n]}(E) \to_\beta$
$(t_{[x,x_1,\ldots,x_n]}(C))\{\!\{\mathtt{1} \leftarrow t_{[x_1,\ldots,x_n]}(E)\}\!\} \overset{L\,48}{=} t_{[x_1,\ldots,x_n]}(C[x := E]) = t_{[x_1,\ldots,x_n]}(B)$ ⊠

Corollary 50 *Let $A, B \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1,\ldots x_n\}$ and $A \twoheadrightarrow_\beta B$.
Then $t_{[x_1,\ldots,x_n]}(A) \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(B)$.*

Lemma 51 *Let $A \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1,\ldots,x_n\}$ and let $y_1,\ldots,y_m$ such that for every i, $1 \le i \le m$, either $y_i \notin FV(A)$ or $y_i = x_j$ for some j, $1 \le j \le n$.
Then $t_{[x_1,\ldots,x_n,y_1,\ldots,y_m]}(A) = t_{[x_1,\ldots,x_n]}(A)$.*

Proof: Easy induction on A. ⊠

Lemma 52 *Let A, $B \in \mathcal{T}$ such that $FV(A) \cup FV(B) \subseteq \{x_1,\ldots,x_n\}$ and $A =_\beta B$.
Then $t_{[x_1,\ldots,x_n]}(A) =_\beta t_{[x_1,\ldots,x_n]}(B)$.*

Proof: By induction on the number k of β-contractions and β-expansions in $A =_\beta B$.
If $k = 0$ the lemma is obvious. There are two possibilities for $k > 0$:

$A =_\beta C \to_\beta B$ **:** If $FV(C) \subseteq \{x_1, \ldots, x_n\}$ apply IH and Lemma 49.
Otherwise let $\{y_1, \ldots, y_m\} = FV(C) \setminus \{x_1, \ldots, x_n\}$. Then

$$t_{[x_1,\ldots,x_n,y_1,\ldots,y_m]}(A) \overset{IH}{=_\beta} t_{[x_1,\ldots,x_n,y_1,\ldots,y_m]}(C) \overset{L49}{\to_\beta} t_{[x_1,\ldots,x_n,y_1,\ldots,y_m]}(B)$$

And we conclude by the previous lemma.

$B \to_\beta C =_\beta A$ **:** By Lemma 19, $FV(C) \subseteq FV(B)$. Now, IH and Lemma 49 settle this case. ⊠

Now we are ready to state and prove the preservation of type assignment by t:

Theorem 53 *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ such that $\Gamma \vdash A : B$.*
Then $t(\Gamma) \vdash t_{[x_n,\ldots,x_1]}(A) : t_{[x_n,\ldots,x_1]}(B)$.

Proof: By induction on the derivation $\Gamma \vdash A : B$. If $A : B$ is an axiom the theorem follows immediately. Let us study the last rule in the derivation $\Gamma \vdash A : B$.

(start) **:** Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$. If $\Gamma \vdash A : s$ then, by IH, $t(\Gamma) \vdash t_{[x_n,\ldots,x_1]}(A) : s$ and hence, applying *(start)*, $t_{[x_n,\ldots,x_1]}(A), t(\Gamma) \vdash \mathtt{1} : U_0^2(t_{[x_n,\ldots,x_1]}(A))$. Since, by definition, $t(\Gamma, x : A) = t_{[x_n,\ldots,x_1]}(A), t(\Gamma)$ and $t_{[x,x_n,\ldots,x_1]}(x) = \mathtt{1}$, we must only check that $t_{[x,x_n,\ldots,x_1]}(A) = U_0^2(t_{[x_n,\ldots,x_1]}(A))$, but this is an instance of Lemma 47, and we are done.

(weakening) **:** As in the previous item, Lemma 47 solves this case.

(application) **:** In this case use Lemma 48.

(abstraction) **and** *(product)* **:** Just use the IH.

(conversion) **:** Lemma 52 settles this case. ⊠

5.2 Translating T to $\mathcal{T}$

Definition 54 (The translation u) *Let $A \in T$ such that $FV(A) \subseteq \{1, \ldots, n\}$ and let $x_1, \ldots, x_n$ be distinct variables of V. We define $u_{[x_n,\ldots,x_1]}(A)$ by induction on A:*

$$\begin{array}{ll}
u_{[x_n,\ldots,x_1]}(c) & =_{def} c \quad \textit{for} \;\; c \in \mathcal{C} \\
u_{[x_n,\ldots,x_1]}(\mathtt{i}) & =_{def} x_i \\
u_{[x_n,\ldots,x_1]}(A\,B) & =_{def} u_{[x_n,\ldots,x_1]}(A) u_{[x_n,\ldots,x_1]}(B) \\
u_{[x_n,\ldots,x_1]}(\pi B.A) & =_{def} \pi x : u_{[x_n,\ldots,x_1]}(B).u_{[x_n,\ldots,x_1,x]}(A) \quad \textit{with } x \notin \{x_1, \ldots, x_n\}
\end{array}$$

Remark that Definition 54 is correct since $FV(\pi B.A) \subseteq \{1, \ldots, n\}$ implies $FV(A) \subseteq \{1, \ldots, n+1\}$. Furthermore, the definition for abstractions and products does not depend on the choice of the variable x thanks to the following remark.

Remark 55 *Let $B, C \in T$ such that $FV(\lambda B.C) \subseteq \{1, \ldots, n\}$, let $x_1, \ldots, x_n$ distinct variables and x, y variables such that x, $y \notin \{x_1, \ldots, x_n\}$.*
Then $\pi x : u_{[x_n,\ldots,x_1]}(B).u_{[x_n,\ldots,x_1,x]}(C) = \pi y : u_{[x_n,\ldots,x_1]}(B).u_{[x_n,\ldots,x_1,y]}(C)$.

Proof: It is enough to show that $u_{[x_n,\ldots,x_1,x]}(C)[x := y] = u_{[x_n,\ldots,x_1,y]}(C)$, which is a particular case of the following lemma. ⊠

Lemma 56 *Let $B \in T$ such that $FV(B) \subseteq \{1, \ldots, n+m+1\}$, and let the variables $x_1, \ldots, x_n$, $z_1, \ldots, z_m$, x and y be all distinct . Then $(u_{[x_n,\ldots,x_1,x,z_m,\ldots,z_1]}(B))[x := y] = u_{[x_n,\ldots,x_1,y,z_m,\ldots,z_1]}(B)$.*

Proof: By induction on B. The only interesting case is when $B = \lambda A.C$. Let $u_{[\underline{x}_n,x,\underline{z}_m]}(B) = \lambda w : u_{[\underline{x}_n,x,\underline{z}_m]}(A).u_{[\underline{x}_n,x,\underline{z}_m,w]}(C)$. Let $u_{[\underline{x}_n,y,\underline{z}_m]}(B) = \lambda v : u_{[\underline{x}_n,y,\underline{z}_m]}(A).u_{[\underline{x}_n,y,\underline{z}_m,v]}(C)$.
Remark that we can assume that $w \neq y$. In fact, if $w = y$ we can choose z such that $z \neq y$ and also distinct from $x_1, \ldots, x_n$, $z_1, \ldots, z_m$, x, and we have
$u_{[\underline{x}_n,x,\underline{z}_m]}(B) = \lambda z : u_{[\underline{x}_n,x,\underline{z}_m]}(A).u_{[\underline{x}_n,x,\underline{z}_m,w]}(C)[w := z] \overset{IH}{=} \lambda z : u_{[\underline{x}_n,x,\underline{z}_m]}(A).u_{[\underline{x}_n,x,\underline{z}_m,z]}(C)$.
Therefore, since $w \neq y$, we have
$(u_{[\underline{x},x,\underline{z}_m]}(B))[x := y] = (\lambda w : u_{[\underline{x}_n,x,\underline{z}_m]}(A).u_{[\underline{x}_n,x,\underline{z}_m,w]}(C))[x := y] =$
$\lambda w : u_{[\underline{x}_n,x,\underline{z}_m]}(A)[x := y].u_{[\underline{x}_n,x,\underline{z}_m,w]}(C)[x := y] \overset{IH}{=} \lambda w : u_{[\underline{x}_n,y,\underline{z}_m]}(A).u_{[\underline{x}_n,y,\underline{z}_m,w]}(C) =$
$\lambda v : u_{[\underline{x}_n,y,\underline{z}_m]}(A).u_{[\underline{x}_n,y,\underline{z}_m,w]}(C)[w := v] \overset{IH}{=} \lambda v : u_{[\underline{x}_n,y,\underline{z}_m]}(A).u_{[\underline{x}_n,y,\underline{z}_m,v]}(C) = u_{[\underline{x}_n,y,\underline{z}_m]}(B)$ ⊠

Definition 57 *Let $\Gamma = A_1, \ldots, A_n$ be a legal context. We define:*
$u(\Gamma) = v_1 : u_{[]}(A_n), v_2 : u_{[v_1]}(A_{n-1}), \ldots, v_n : u_{[v_1,\ldots,v_{n-1}]}(A_1)$

Definition 57 is correct thanks to Lemma 43.

As in the previous subsection, we must establish some lemmas in order to prove that u preserves type assignement. These lemmas clarify the interaction of u with the updating functions, substitutions, β-contractions and β-equivalence.

Lemma 58 *Let $A \in \mathcal{T}$, $i \geq 1$, $k \geq 0$ and $n \geq k+i$ such that $FV(A) \subseteq \{1, \ldots, n-i+1\}$. Then $u_{[x_n,\ldots,x_1]}(U_k^i(A)) = u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1]}(A)$.*

Proof: By induction on A. As usual we study the two interesting cases:
$A = \mathtt{m}$ **:** If $m \leq k$ then $u_{[x_n,\ldots,x_1]}(U_k^i(A)) = x_m = u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1]}(A)$.
If $m > k$ then $u_{[x_n,\ldots,x_1]}(U_k^i(A)) = x_{m+i-1} = u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1]}(A)$.
$A = \lambda B.C$ **:** We can choose x to obtain:
$u_{[x_n,\ldots,x_1]}(U_k^i(A)) = \lambda x : u_{[x_n,\ldots,x_1]}(U_k^i(B)).u_{[x_n,\ldots,x_1,x]}(U_{k+1}^i(C)) \overset{IH}{=}$
$\lambda x : u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1]}(B).u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1,x]}(C) = u_{[x_n,\ldots,x_{k+i},x_k,\ldots,x_1]}(A)$ ⊠

Lemma 59 *Let A, $B \in \mathcal{T}$ and $x_1, \ldots, x_n$, $y_1, \ldots, y_{i-1}$, x distinct variables.*
Then $u_{[x_n,\ldots,x_1,y_{i-1},\ldots,y_1]}(A\{\!\{\mathtt{i} \leftarrow B\}\!\}) = (u_{[x_n,\ldots,x_1,x,y_{i-1},\ldots,y_1]}(A))[x := u_{[x_n,\ldots,x_1]}(B)]$.

Proof: By induction on A. We study the two interesting cases:
$A = \mathtt{j}$ **:** If $j < i$ then $u_{[\underline{x}_n,\underline{y}_{i-1}]}(A\{\!\{\mathtt{i} \leftarrow B\}\!\}) = y_j = (u_{[\underline{x}_n,x,\underline{y}_{i-1}]}(A))[x := u_{[x_n,\ldots,x_1]}(B)]$.
If $j > i$ then $u_{[\underline{x}_n,\underline{y}_{i-1}]}(A\{\!\{\mathtt{i} \leftarrow B\}\!\}) = x_{j-i} = (u_{[\underline{x}_n,x,\underline{y}_{i-1}]}(A))[x := u_{[x_n,\ldots,x_1]}(B)]$.
If $j = i$ then $u_{[\underline{x}_n,\underline{y}_{i-1}]}(A\{\!\{\mathtt{i} \leftarrow B\}\!\}) = u_{[\underline{x}_n,\underline{y}_{i-1}]}(U_0^i(B)) \overset{L58}{=}$
$u_{[x_n,\ldots,x_1]}(B) = (u_{[\underline{x}_n,x,\underline{y}_{i-1}]}(A))[x := u_{[x_n,\ldots,x_1]}(B)]$.
$A = \lambda D.C$ **:** We choose $z \neq x$ to obtain:
$u_{[\underline{x}_n,\overline{y}]}(A\{\!\{\mathtt{i} \leftarrow B\}\!\}) = \lambda z : u_{[\underline{x}_n,\underline{y}_{i-1}]}(D\{\!\{\mathtt{i} \leftarrow B\}\!\}).u_{[\underline{x}_n,\underline{y}_{i-1},z]}(C\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) \overset{IH}{=}$
$\lambda z : (u_{[\underline{x}_n,x,\underline{y}_{i-1}]}(D))[x := u_{[x_n,\ldots,x_1]}(B)].(u_{[\underline{x}_n,x,\underline{y}_{i-1},z]}(C))[x := u_{[x_n,\ldots,x_1]}(B)] =$
$(u_{[\underline{x}_n,x,\underline{y}_{i-1}]}(A))[x := u_{[x_n,\ldots,x_1]}(B)]$ ⊠

Lemma 60 *Let $A, B \in \mathcal{T}$ such that $FV(A) \subseteq \{1, \ldots n\}$ and $A \to_\beta B$.*
Then $u_{[x_n,\ldots,x_1]}(A) \to_\beta u_{[x_n,\ldots,x_1]}(B)$.

Proof: Remark that Lemma 39 guarantees the good definition of $u_{[x_n,\ldots,x_1]}(B)$.
The proof is by induction on A. The interesting case is when A is an application and the reduction takes place at the root.
Therefore, let $A = (\lambda D.C)E$ and $B = C\{\!\{\mathtt{1} \leftarrow E\}\!\}$. We have
$u_{[x_n,\ldots,x_1]}(A) = (\lambda x : u_{[x_n,\ldots,x_1]}(D).u_{[x_n,\ldots,x_1,x]}(C))u_{[x_n,\ldots,x_1]}(E) \to_\beta$
$(u_{[x_n,\ldots,x_1,x]}(C))[x := u_{[x_n,\ldots,x_1]}(E)] \overset{L59}{=} u_{[x_n,\ldots,x_1]}(C\{\!\{\mathtt{1} \leftarrow E\}\!\}) = u_{[x_n,\ldots,x_1]}(B)$ ⊠

Corollary 61 *Let $A, B \in \mathcal{T}$ such that $FV(A) \subseteq \{1, \ldots n\}$ and $A \twoheadrightarrow_\beta B$.*
Then $u_{[x_n,\ldots,x_1]}(A) \twoheadrightarrow_\beta u_{[x_n,\ldots,x_1]}(B)$.

Lemma 62 *Let $A \in \mathcal{T}$ such that $FV(A) \subseteq \{1, \ldots, n\}$ and let $m \geq n$.*
Then $u_{[x_m,\ldots,x_1]}(A) = u_{[x_n,\ldots,x_1]}(A)$.

Proof: Easy induction on A. ⊠

Lemma 63 *Let A, $B \in \mathcal{T}$ such that $FV(A) \cup FV(B) \subseteq \{1, \ldots, n\}$ and $A =_\beta B$.*
Then $u_{[x_n,\ldots,x_1]}(A) =_\beta u_{[x_n,\ldots,x_1]}(B)$.

Proof: By induction on the number of β-contractions and β-expansions in $A =_\beta B$. The proof is analogous to the proof of Lemma 52 and uses Lemma 62. ⊠

Here is now the preservation of type assignment by u:

Theorem 64 *If $\Gamma \vdash A : B$ then $u(\Gamma) \vdash u_{[v_1,\ldots,v_n]}(A) : u_{[v_1,\ldots,v_n]}(B)$.*

Proof: By induction on the derivation $\Gamma \vdash A : B$. If $A : B$ is an axiom the theorem follows immediately. Let us study the last rule in the derivation $\Gamma \vdash A : B$.

(start) **:** Let $\Gamma = A_1, \ldots, A_n$. If $\Gamma \vdash A : s$ then, by IH, $u(\Gamma) \vdash u_{[v_1,\ldots,v_n]}(A) : s$ and hence, applying *(start)*, $u(\Gamma), v_{n+1} : u_{[v_1,\ldots,v_n]}(A) \vdash v_{n+1} : u_{[v_1,\ldots,v_n]}(A)$. Since, by Definition 57, $u(A,\Gamma) = u(\Gamma), v_{n+1} : u_{[v_1,\ldots,v_n]}(A)$ and $u_{[v_1,\ldots,v_{n+1}]}(\mathtt{1}) = v_{n+1}$, we must only check that $u_{[v_1,\ldots,v_{n+1}]}(U_0^2(A)) = u_{[v_1,\ldots,v_n]}(A)$, but this is an instance of Lemma 58, and we are done.

(weakening) **:** As in the previous item, Lemma 58 solves this case.

(application) **:** For this case use Lemma 59.

(abstraction) **and** *(product)* **:** Just use the IH.

(conversion) **:** Lemma 63 settles this case. ⊠

5.3 t and u are inverses

We must check that the compositions of t and u are the identity. We begin by studying $t \circ u$, which as expected is exactly the identity. We prove first the following lemma:

Lemma 65 *Let $A \in T$ such that $FV(A) \subseteq \{1, \ldots, n\}$ and let $x_1, \ldots, x_n$ be distinct variables. Then $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(A)) = A$.*

Proof: By induction on A. The usual two interesting cases are:

$A = \mathtt{i}$ **:** Since $x_1, \ldots, x_n$ are distinct variables, we have:
$t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(A)) = t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(\mathtt{i})) = t_{[x_1,\ldots,x_n]}(x_i) = \mathtt{i} = A$

$A = \lambda B.C$ **:** We have: $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(A)) = t_{[x_1,\ldots,x_n]}(\lambda x : u_{[x_n,\ldots,x_1]}(B).u_{[x_n,\ldots,x_1,x]}(C)) =$
$\lambda t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(B)).t_{[x,x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1,x]}(C)) \overset{IH}{=} \lambda B.C$ ⊠

We use $\vdash_T$ for type derivations in PTSs with de Bruijn indices, and $\vdash_{\mathcal{T}}$ for type derivations in PTSs with variable names.

Definition 66 (Derivations that are exactly the same) *We say that two derivations $\Gamma \vdash A : B$ and $\Gamma' \vdash A' : B'$ are exactly the same if $\Gamma = \Gamma$, $A = A'$ and $B = B'$.*

Proposition 67 *Let $\Gamma = A_1, \ldots, A_n$ such that $\Gamma \vdash A : B$. Then the derivations $\Gamma \vdash_T A : B$ and $t(u(\Gamma)) \vdash_T t_{[v_n,\ldots,v_1]}(u_{[v_1,\ldots,v_n]}(A)) : t_{[v_n,\ldots,v_1]}(u_{[v_1,\ldots,v_n]}(B))$ are exactly the same.*

Proof: Remark that $t(u(\Gamma)) = t(v_1 : u_{[]}(A_n), v_2 : u_{[v_1]}(A_{n-1}), \ldots, v_n : u_{[v_1,\ldots,v_{n-1}]}(A_1)) =$
$t_{[v_{n-1},\ldots,v_1]}(u_{[v_1,\ldots,v_{n-1}]}(A_1)), \ldots, t_{[v_1]}(u_{[v_1]}(A_{n-1})), t_{[]}(u_{[]}(A_n)) \overset{L\,65}{=} \Gamma$. Using again Lemma 65 we are done. ⊠

We study now $u \circ t$. We cannot expect to have exactly the identity now, since when we translate de Bruijn derivations we choose the variables in the declarations of the context in a determined way: v_1, v_2, etc. Therefore we are going to end up with a derivation which differs from the original one in the choice of these variables. We say that these derivations are *equivalent* and this notion of equivalence is defined precisely as follows:

Definition 68 (Equivalent derivations) *For any context Γ and any term $A \in \mathcal{T}$ we define $\pi\Gamma.A$, for $\pi \in \{\Pi, \lambda\}$ by induction on the length of the context as follows:*
$\pi <> .A =_{def} A$ *and* $\pi(\Gamma, x : B).A =_{def} \pi\Gamma.\pi x : B.A$

We say that the derivations $\Gamma \vdash_{\mathcal{T}} A : B$ and $\Gamma' \vdash_{\mathcal{T}} A' : B'$ are equivalent *when $\lambda\Gamma.A =_\alpha \lambda\Gamma'.A'$ and $\Pi\Gamma.B =_\alpha \Pi\Gamma'.B'$.*

Lemma 69 *Let $A \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1, \ldots, x_n\}$ and $x_1, \ldots, x_n$ are distinct variables. Then $u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A)) =_\alpha A$.*

Proof: By induction on A. The usual two interesting cases are:
$A = x_i$ **:** Since $x_1, \ldots, x_n$ are distinct variables, we have:
$u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A)) = u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(x_i)) = u_{[x_n,\ldots,x_1]}(\mathtt{i}) = x_i = A$
$A = \lambda x : B.b$ **:** By VC we can assume x distinct from $x_1, \ldots, x_n$. We have:
$u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A)) = u_{[x_n,\ldots,x_1]}(\lambda t_{[x_1,\ldots,x_n]}(B).t_{[x,x_1,\ldots,x_n]}(b) =$
$\lambda x : u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(B)).u_{[x_n,\ldots,x_1,x]}(t_{[x,x_1,\ldots,x_n]}(b)) \overset{IH}{=}_\alpha \lambda x : B.b$ ⊠

Proposition 70 *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $A, B \in \mathcal{T}$. The derivations $\Gamma \vdash_{\mathcal{T}} A : B$ and $u(t(\Gamma)) \vdash_{\mathcal{T}} u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(A)) : u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(B))$ are equivalent in the sense of Definition 68.*

Proof: By induction on the length of Γ. For $\Gamma = \emptyset$ we use Lemma 69. Therefore let us assume that the proposition holds for $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and let us prove it for the context $\Gamma, x_{n+1} : A_{n+1}$.
Hence, we must establish the following α-congruences:

$$\lambda u(t(\Gamma, x_{n+1} : A_{n+1})).u_{[v_1,\ldots,v_{n+1}]}(t_{[x_{n+1},\ldots,x_1]}(A)) =_\alpha \lambda(\Gamma, x_{n+1} : A_{n+1}).A$$

$$\Pi u(t(\Gamma, x_{n+1} : A_{n+1})).u_{[v_1,\ldots,v_{n+1}]}(t_{[x_{n+1},\ldots,x_1]}(B)) =_\alpha \Pi(\Gamma, x_{n+1} : A_{n+1}).B$$

We prove the first one. Note that $u(t(\Gamma, x_{n+1} : A_{n+1})) = u(t(\Gamma)), v_{n+1} : u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(A_{n+1}))$.
Therefore, $\lambda u(t(\Gamma, x_{n+1} : A_{n+1})).u_{[v_1,\ldots,v_{n+1}]}(t_{[x_{n+1},\ldots,x_1]}(A)) =$
$\lambda u(t(\Gamma)).\lambda v_{n+1} : u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(A_{n+1})).u_{[v_1,\ldots,v_{n+1}]}(t_{[x_{n+1},\ldots,x_1]}(A)) =$
$\lambda u(t(\Gamma)).u_{[v_1,\ldots,v_n]}(\lambda t_{[x_n,\ldots,x_1]}(A_{n+1}).t_{[x_{n+1},\ldots,x_1]}(A)) =$
$\lambda u(t(\Gamma)).u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(\lambda x_{n+1} : A_{n+1}.A)) \overset{IH}{=}_\alpha$
$\lambda \Gamma.\lambda x_{n+1} : A_{n+1}.A = \lambda(\Gamma, x_{n+1} : A_{n+1}).A$ ⊠

The following two lemmas establish that both t and u preserve strong normalisation in the following sense:

Lemma 71 *Let $A \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1, \ldots, x_n\}$. If A is strongly normalising then so is $t_{[x_1,\ldots,x_n]}(A)$.*

Proof: Assume that $t_{[x_1,\ldots,x_n]}(A)$ is not SN. I.e., there is an infinite sequence of A_i, $i \geq 1$ such that $t_{[x_1,\ldots,x_n]}(A) \to_\beta A_1 \to_\beta A_2 \ldots$. By Lemma 46, $FV(t_{[x_1,\ldots,x_n]}(A)) \subseteq \{1, \ldots, n\}$ and hence by Lemma 39, $FV(A_i) \subseteq \{1, \ldots, n\}$ for $i \geq 1$. By Lemma 60, $u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A)) \to_\beta u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A_1)) \to_\beta (u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A_2)) \ldots$. By Lemma 69, $A \to_\beta A_1 \to_\beta A_2 \ldots$ and hence A is not SN. This is absurd. ⊠

Lemma 72 *Let $A \in T$ such that $FV(A) \subseteq \{1, \ldots, n\}$. If A is strongly normalising then so is $u_{[x_n,\ldots,x_1]}(A)$.*

Proof: Similar to the proof of Lemma 71 ⊠

6 Properties of PTSs with de Bruijn indices

In this section, we will establish the properties of the PTSs with variable names listed in Section 3.2 for our formulation of the PTSs with de Bruijn indices. First, note that Lemmas 19 and 21 have already been established for de Bruijn indices in Lemmas 39 and 43 respectively. Theorems 20, 22 and 25 will be established as Theorems 73, 74 and 76 below and Lemma 24 will be established as Lemma 75 below. First, note that the definition of singly sorted PTSs (Definition 23) is unchanged for de Bruijn indices. Moreover, the notion singly sorted does not get lost during translation between variable names and de Bruijn indices.

Theorem 73 (The Church Rosser Theorem for PTSs with de Bruijn indices) *In any PTS with de Bruijn indices we have:*
if $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$ then there exists D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.

Proof: Assume $FV(A) \subseteq \{1, \ldots, n\}$ and let $x_1, \ldots, x_n$ be distinct variables of V. By Corollary 61, $u_{[x_n,\ldots,x_1]}(A) \twoheadrightarrow_\beta u_{[x_n,\ldots,x_1]}(B)$ and $u_{[x_n,\ldots,x_1]}(A) \twoheadrightarrow_\beta u_{[x_n,\ldots,x_1]}(C)$. Hence, by Theorem 20, $\exists D$ such that $u_{[x_n,\ldots,x_1]}(B) \twoheadrightarrow_\beta D$ and $u_{[x_n,\ldots,x_1]}(C) \twoheadrightarrow_\beta D$. Note that $FV(u_{[x_n,\ldots,x_1]}(B)) \subseteq \{x_1, \ldots, x_n\}$ and $FV(u_{[x_n,\ldots,x_1]}(C)) \subseteq \{x_1, \ldots, x_n\}$ and hence by Corollary 50 we have: $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(B)) \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$ and $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(C)) \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$. Then, Corollary 40 sorts out the free variable condition for Lemma 65, and the latter gives $B \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$ and $C \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$. ⊠

Theorem 74 (Subject Reduction SR, for PTSs with de Bruijn indices)
If $\Gamma \vdash_T A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash_T A' : B$.

Proof: First, we use Theorem 64 and Lemma 60 to obtain the conditions of Theorem 22 in $\mathcal{T}$. Then, we use Theorem 53 and Proposition 67 to obtain SR in T. ⊠

Lemma 75 (Unicity of types for singly sorted PTSs with de Bruijn indices)
In any singly sorted PTS, the following holds:

1. *If $\Gamma \vdash_T A : B_1$ and $\Gamma \vdash_T A : B_2$ then $B_1 =_\beta B_2$.*
2. *If $\Gamma \vdash_T A : B$ and $\Gamma \vdash_T A' : B'$ then $A =_\beta A'$ implies $B =_\beta B'$.*
3. *If $\Gamma \vdash_T B : S$, $B =_\beta B'$ and $\Gamma \vdash_T A' : B'$, then $\Gamma \vdash_T B' : S$.*

Proof: We will only show 1. The other two cases are similar. Assume that $\Gamma \vdash_T A : B_1$ and $\Gamma \vdash_T A : B_2$. Then, by Theorem 64, $u(\Gamma) \vdash_{\mathcal{T}} u_{[x_n,\ldots,x_1]}(A) : u_{[x_n,\ldots,x_1]}(B_1)$ and $u(\Gamma) \vdash_{\mathcal{T}} u_{[x_n,\ldots,x_1]}(A) : u_{[x_n,\ldots,x_1]}(B_2)$. By Lemma 24 we get $u_{[x_n,\ldots,x_1]}(B_1) =_\beta u_{[x_n,\ldots,x_1]}(B_2)$. Finally, by Lemma 52, $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(B_1)) =_\beta t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(B_2))$ and we use Lemma 65 to get $B_1 =_\beta B_2$. ⊠

Theorem 76 (Strong normalisation for the systems of the cube) *Every legal term of the cube with de Bruijn indices is strongly normalising.*

Proof: Let $\Gamma \vdash_T A : B$ and let us show that A and B are SN. By Theorem 64, $u(\Gamma) \vdash_{\mathcal{T}} u_{[x_n,\ldots,x_1]}(A) : u_{[x_n,\ldots,x_1]}(B)$. Hence, $u_{[x_n,\ldots,x_1]}(A)$ and $u_{[x_n,\ldots,x_1]}(B)$ are SN by Theorem 25. Now by Lemma 71, $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(A))$ and $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(B))$ are SN and so by Lemma 65, A and B are SN and we are done. ⊠

7 Conclusions

Although type theory and the λ-calculus are vital for the foundations and implementation of programming languages, they are usually written using variable names which are problematic to implement. For this reason, during implementations, variable names are replaced by notions such as combinators, de Bruijn indices, graphs, and so on. PTSs like other type systems have been introduced using variable names. This paper provides a formulation of PTSs using de Bruijn indices and establishes an isomorphism between this formulation and the one with variable names. This isomorphism is then used to establish the properties of PTSs with de Bruijn indices.

As we said above, de Bruijn indices are usually used in implementations of type theory and of the λ-calculus. However, when proving properties of these implementations, it is important to know what are the properties of the systems written with de Bruijn indices and used during the implementation. This is the reason for the interest in establishing results such as confluence for the type-free λ-calculus with de Bruijn indices and its isomorphism to the type-free λ-calculus with variable names. Another example we give for this is the formulation of some typed λ-calculi with de Bruijn indices in order to build systems of explicit substitutions (e.g., see [1]). As far as we know, our work is the first formulation of PTSs with de Bruijn indices. For this formulation, we establish its isomorphism to PTSs with variable names, and its desirable properties such as confluence, subject reduction, unicity of types and strong normalisation (when appropriate), etc.

Of course this formulation opens the door to future work where extensions of type theory can be written using either named variables or de Bruijn's indices. We have already for instance started writing generalised reductions using de Bruijn indices (cf. [15]) and this enabled us to combine generalised reduction with explicit substitutions to obtain a system where there is more control in delaying computations. Explicit substitutions is another area where avoiding α-conversion is desirable because the main goal of explicit substitutions is to improve and control computations. De Bruijn's indices are one way of avoiding α-conversion and hence our work on formulating PTSs with de Bruijn indices opens the door to extensions of these PTSs with explicit substitutions in a useful way for implementations. There is yet no extensions of PTSs (containing all the systems of the cube) with explicit substitutions using de Bruijn indices and satisfying the desirable properties. We are investigating this point at the moment.

Another useful extension of PTSs is adding definitions (or let expressions) and/or Π-reductions [12] or other forms of generalised reductions [15]. We have worked these extensions for variable names and we plan to conduct this work for de Bruijn indices.

Finally, in this paper, we rewrote PTSs using de Bruijn indices. It is interesting to do the same using combinators and/or other disciplines of treating/avoiding variables which has not yet been used in PTSs. Our next goal is to investigate PTSs with combinators.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science, Volume 2: Background: Computational Structures.* Oxford University Press, 1992.
3. H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.
4. H.P. Barendregt. Lambda calculi with types. In [2], pages 117–309. Oxford University Press, 1992.
5. S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Dept. of Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, 1988.
6. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the church rosser theorem. In *Proceedings of the Koninklijke Nederlandse Academie van Wetenschappen.* Mathematical Sciences, 1972. Volume 75; also in [18], Chapter C2.
7. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.
8. T. Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, Thèse de troisième cycle, 1985.
9. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
10. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings Second Symposium on Logic in Computer Science*, pages 194–204, Washington D.C., 1987. IEEE.
11. J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.
12. F. Kamareddine, R. Bloo, and R.P. Nederpelt. On π-conversion in the λ-cube and the combination with abbreviations. *Annals of Pure and Applied Logics*, 97:27–45, 1999.
13. F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. Proceedings of 7th international symposium on Programming Languages: Implementations, Logics and Programs, PLILP'95. *Lecture Notes in Computer Science 982*, pages 45–62, 1995.
14. F. Kamareddine and A. Ríos. Bridging de Bruijn indices and variable names in explicit subst itutions calculi. *The Logic Journal of the Interest Group of Pure and Applied Logic, IGPL*, 6(6):843–874, 1998.
15. F. Kamareddine, A. Ríos, and J.B. Wells. Calculi of Generalised β-reduction and explicit substitution: Type Free and Simply Typed Versions. *Journal of Functional and Logic Programming*, 1998(Article 5):1–44, 1998.

16. J. McKinna and R. Pollack. Pure type systems formalised. In M. Bezem and J.-F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 289–305, 1993.
17. R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML.* MIT Press, Cambridge (Massachusetts)/London, 1990.
18. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath.* Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam, 1994.
19. F. Pfenning. A proof of the church rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Carnegie Mellon University, 1992.
20. K.H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, University of Copenhagen, 1996.
21. J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.