Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems

FAIROUZ KAMAREDDINE, *Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, Scotland.*
E-mail: fairouz@cee.hw.ac.uk

Abstract

This article is a brief review of the type-free λ-calculus and its basic rewriting notions, and of the pure type system framework which generalises many type systems. Both the type-free λ-calculus and the pure type systems are presented using variable names and de Bruijn indices. Using the presentation of the λ-calculus with de Bruijn indices, we illustrate how a calculus of explicit substitutions can be obtained. In addition, de Bruijn's notation for the λ-calculus is introduced and some of its advantages are outlined.

Keywords: Types, rewriting, λ-calculus and de Bruijn's notation.

1 Introduction to logics, types and rewriting

Logic has existed since ancient times, really it goes back to the consciousness of human beings. However, in the 20th century, there has been an explosion in the different logics introduced and in the applications that depended on logic. This explosion is due to many reasons that we will briefly touch on in this paper. This explosion moreover, is not slowing down in the twenty-first century. We will continue to see new different logics, extensions of old logics, and the study of their theory and applications will thrive as it did in the last century. This is not surprising because the twentieth century was indeed a *century of complexity* and this complexity will be carried to this century. The following table explains the consequences of a single machine failure in the years 1900 and in 2000.

	1900	2000
Main way information travels in society:	paper	electric signals, radio
Number of parts in complex machine:	10,000 (locomotive)	1,000,000,000 (CPU)
Worst consequences of single machine failure:	100s die	end of all life?
Likelihood a machine includes a computer:	very low	very high

This complexity of information and the disastrous consequences of failure, lead to the need for **Automation** and for establishing **Correctness**. Modern technological systems are just too complicated for humans to reason about unaided, so *automation* is needed. In addition, because of the increasing interdependency of systems and the faster and more automatic travel of information, failures can have a wide impact. So establishing *correctness* is important.

© Oxford University Press

Furthermore, because modern systems have so many possible states, *testing* is often impractical. It seems that *proofs* are needed to cover infinitely many situations. In other words, *some* kind of formalism is needed to *aid in design* and to *ensure safety*.

But then, *what kind of formalism* should one develop? This is not an easy question to answer. However, a reasoning formalism should *at least* be:

- **Correct**: Only correct statements can be 'proven'.
- **Adequate**: Necessary properties in the problem domain can be stated and proved.
- **Feasible**: The resources (money, time) used in stating and proving the needed properties must be within practical limits.

In addition, assuming a minimally acceptable formalism, we would also like it to be:

- **Efficient**: Costs of both the reasoning process *and* the thing being reasoned about should be minimized.
- **Supportive of reuse**: Slight specification changes should not force reproving properties for an entire system. Libraries of pre-proved statements should be well supported.
- **Elegant**: The core of the reasoning formalism should be as simple as possible, to aid in reasoning about the formalism itself.

Work on logic in the twentieth century led to the development of two related yet complementary areas: *types* and *rewriting*. Logics, types, and rewriting have existed in various forms since the times of the ancient Babylonians and Greeks (e.g. Euclid, Aristotle, etc.) yet in the twentieth century, types and rewriting became *explicit* theories and started to be developed as branches of their own. Logics, types, and rewriting are able to be shown *correct*, are *elegant* as we can formulate and (automate) clear rules of how they work (e.g. from A and $A \to B$ we can deduce B), and are *adequate* as we can express a lot in these tiny formalisms.

But what are logics? What are proofs? What are types and what is rewriting? Here is an attempt at explaining them. We start with proofs and logic.

- A proof is the **guarantee** of some statement provided by a rigorous **explanation** stated using **axioms** (statements 'for free') and **rules** for combining already proven statements to obtain more statements.
- A logic is a formalism for statements and proofs of statements.
- Why do we believe the explanation of a proof? Because a proved statement is derived step by step from explicit assumptions using a trusted logic.

The above explanation of logic and proofs can be traced back to the times of Aristotle (384–322 BC) who wanted a set of rules that would be powerful enough for most intuitively valid proofs. Aristotle correctly stated that **proof search** is harder than **proof checking**:

> Given a proof of a statement, one can check that it is a correct proof. Given a statement, one may not be able to find the proof.

Aristotle's intuitions on this have been confirmed by Gödel, Turing, and others in the twentieth century. Much later than Aristotle, Leibniz (1646–1717) conceived of **automated deduction**, i.e. to find

- a language L in which arbitrary concepts could be formulated, and
- a machine to determine the correctness of statements in L.

Such a machine cannot work for every statement according to Aristotle and (later results by) Gödel and Turing.

The late 1800s saw the beginnings of serious formalization: Cantor began formalizing set theory [6, 7] and made contributions to number theory, Peano formalized arithmetic [39], Frege's *Begriffsschrift* [13] (1879) gave the first thorough and extensive formalization of logic. Frege's *Grundgesetze der Arithmetik* [14, 16], called later by others Naive Set Theory (NST), could handle elementary arithmetic, set theory, logic, and quantification. Frege's NST allowed a precise definition of the vital concept of the **function**. As a result, NST could include not only functions that take numbers as arguments and return numbers as results, but also functions that can take and return other sorts of arguments, *including functions*. These powerful functions were the key to the formalization of logic in NST. Frege was cautious: ordinary functions could only take 'objects' as arguments, not other functions. However, to gain important expressive power, he allowed a way to turn a function into an object representing its graph. Unfortunately, this led to a **paradox**, due to the implicit possibility of **self-application** of functions. In 1902, Russell suggested [42] and Frege completed the argument [15] that a **paradox** could occur in NST. First, one can define S to be 'the set of all sets which do not contain themselves'. Then, one can prove *both* of these statements in NST:

$$S \in S \qquad \Leftrightarrow \qquad S \notin S$$

The same paradox could be encoded in the systems of Cantor and Peano (but not in Frege's weaker Begriffsschrift). As a result, all these systems were **inconsistent** — not only could every true statement be proved but also every false one! (Three-valued logic can solve this, but is unsatisfactory for other reasons.) Logic was in a *crisis*.

In 1908, Russell suggested the use of **types** to solve the problem [43]. It is fair to say that types were (implicitly) used much earlier than that. For example, Euclid's *Elements* (circa 325 BC) begins with (see page 153 of [12]):

1. A **point** is that which has no part.
2. A **line** is breadthless length.

   $\vdots$

15. A **circle** is a plane figure contained by one line such that all the straight lines falling upon it from one point among those lying within the figure are equal to one another.

Although the above seems to merely *define* points, lines, and circles, it shows more importantly that Euclid *distinguished* between them. Euclid always mentioned to which **class** (points, lines, etc.) an object belonged. By distinguishing classes of objects, Euclid prevented undesired situations, like considering whether two points (instead of two lines) are parallel. When considering whether two objects were parallel, intuition forced Euclid to think about the *type* of the objects. As intuition does not support the notion of parallel points, he did not even *try* to undertake such a construction.

In this manner, types have always been present in mathematics, although they were not noticed explicitly until the late 1800s. If you have studied geometry, then you have some (implicit) understanding of types. The question that poses itself then is what led to the creation of this new discipline (type theory) in the twnetieth century. Twan Laan in his PhD thesis [33] gives an excellent survey of the evolution of type theory. Here, we briefly use his argument to state that starting in the 1800s, mathematical systems became less intuitive, for several reasons:

- Very complex or abstract systems.
- Formal systems.
- Something with less intuition than a human using the systems: a computer.

These situations are *paradox threats*. An example is Frege's NST. In such cases, there is not enough intuition to activate the (implicit) type theory to warn against an impossible situation. Reasoning proceeds within the impossible situation and then obtains a result that may be wrong or paradoxical.

To avoid the paradoxes of the systems of Cantor, Peano, and Frege, Russell prescribed avoiding self-reference and self-application in his 'vicious circle principle':

Whatever involves all *of a collection must not be one of the collection.*

Russell implemented this in his Ramified Theory of Types (RTT) [43] which used *types* and *orders*. Self-application was prevented by forcing functions of order k to be applied only to arguments of order less than k. This was carried out further by Russell and Whitehead in the famous *Principia Mathematica* [49] (1910–1912), which founded mathematics on logic, as far as possible, avoiding paradoxes. For example, in RTT, one can define a function '$+$' which is restricted to be applied only to integers.

Although RTT was correct, unlike NST, the types of RTT have turned out instead to be *too restrictive* for mathematics and computer science where fixed points (to mention one example) play an important role. RTT also forces duplication of the definitions of the number system, the Boolean algebra, etc., at *every* level.

The exploration of the middle ground between these two extremes has led to many systems, most of them in the context of the λ-calculus, the first higher-order *rewriting* system. If you have studied algebra, then you know some basics in rewriting. Here is an example of algebraic calculations which illustrates how rewriting works:

$$\begin{array}{cll}
 & (\underline{a+b}) - a & \text{by rule} \quad x+y = y+x \\
= & \underline{(b+a) - a} & \text{by rule} \quad x-y = x+(-y) \\
= & \underline{(b+a)+(-a)} & \text{by rule} \quad (x+y)+z = x+(y+z) \\
= & b+(\underline{a+(-a)}) & \text{by rule} \quad x+(-x) = 0 \\
= & \underline{b+0} & \text{by rule} \quad x+0 = x \\
= & b &
\end{array}$$

Rewriting is the action of replacing a subexpression which is matched by an instance of one side of a rule by the corresponding instance of the other side of the same rule. Important properties of rewriting systems include:

- **Orientation**: Usually, most rules can only be used from left to right as in $x + 0 \to x$. Forward use of the oriented rules represents progress in computation. Un-oriented rules usually do trivial work as in $x + y = y + x$.
- **Termination**: It is desirable to show that rewriting halts, i.e. to avoid infinite sequences of the form $P \to P_1 \to P_2 \to \cdots$.
- **Confluence**: The result of rewriting is independent of the order in which the rules are used. For example, $1 + 2 + 3$ should rewrite to 6, no matter how we evaluate it.

As for types, computations (or rewriting) existed since ancient times (e.g. algebra). However, only in the twentieth century, have higher-order rewriting calculi and theories been extensively developed and important themes and problems identified and studied. In this paper, we are only interested in the development of higher-order rewriting through the λ-calculus which was highly influenced by Frege's **abstraction principle** of the late 1800s. This principle states that any expression mentioning some symbol in zero or more places can be turned into a function by abstracting over that symbol. Introduced in the 1930s, Church's λ-calculus made function abstraction an *operator*. For example, $(\lambda x.\, x + 5)$ represents the (unnamed) mathematical function which takes as input any number and returns as output the result of adding 5 to that number. The λ-calculus provides **higher-order** rewriting, allowing equations like:

$$f(\underline{(\lambda x.\, x + (1/x))5}) = f(5 + (\underline{1/5})) = f(\underline{5 + 0.2}) = f(5.2)$$

The type-free λ-calculus, which can be seen as a small programming language, is an excellent theory of functions — it can represent all computable functions. Church intended the *type-free* λ-calculus with logical operators to provide a foundation for mathematics. Unfortunately, Russell's paradox could also be encoded in the type-free λ-calculus, rendering its use for logic incorrect. Church [8] and Curry [11] introduced the simply typed λ-calculus (STLC) to provide logic while avoiding Russell's paradox in a manner similar to RTT. Unfortunately, like RTT, the STLC is too restrictive.

The areas, Logics, Types and Rewriting converge. Heyting [20], Kolmogorov [32], Curry and Feys [11] (improved by Howard [22]), and de Bruijn [38] all observed the '**propositions as types**' or '**proofs as terms**' (PAT) correspondence. In PAT, logical operators are embedded in the types of λ-terms rather than in the propositions and λ-terms are viewed as proofs of the propositions represented by their types. Advantages of PAT include the ability to manipulate proofs, easier support for independent proof checking, the possibility of the extraction of computer programs from proofs, and the ability to prove properties of the logic via the termination of the rewriting system.

In the present time, there is a remarkable revival of λ-calculus, especially in the versions which use types. Both logicians and computer scientists have developed several branches of typed and untyped λ-calculus. Also mathematics has benefitted from λ-calculus, especially since the time (around 1970) where de Bruijn used his λ-calculus-based Automath for the analysis and checking of mathematical texts. In the rest of this article, we give a brief introduction, both in the classical notation of Church and in the de Bruijn notation, to the λ-calculus and to type theory via the *pure type systems* framework. Section 2 deals with the type-free λ-calculus and Section 3 deals with pure type systems. In particular, we introduce in Section 2.1 some basic rewriting notions needed for the λ-calculus and in Section 2.2 we give the classical λ-calculus (as is usually written) with variable names. In Section 2.3 we present the classical λ-calculus with de Bruijn indices and in Section 2.4 we turn it into a calculus of explicit substitutions. In Section 2.5 we present the λ-calculus using variable names in de Bruijn's notation rather than in the classical one. In this presentation, called λ-calculus à la de Bruijn, the argument appears before the function and terms are structured in a different manner to the classical λ-calculus. The λ-calculus à la de Bruijn can also be written using de Bruijn indices instead of variable names, and we refer the reader to [23] for further details. In Section 3.1 we present the pure type systems framework in the classical notation of the λ-calculus using variable names. In Section 3.2 we present the pure type systems in classical notation using de Bruijn indices and establish their isomorphism to the version with variable names. We leave it as an exercise for the reader to write pure type systems in de Bruijn's notation (using either variable names or de Bruijn indices).

2 The type-free λ-calculus

In this section, we introduce the *classical* λ-calculus (with variable names and with de Bruijn indices) and the λ-calculus *à la de Bruijn*. Terms of the classical λ-calculus are constructed via application (as in AB) or abstraction (as in $\lambda v.A$ if variable names are used, or λA if de Bruijn indices are used). Terms of the λ-calculus à la de Bruijn are also constructed using application (as in $(B)A$) or abstraction (as in $[v]A$ if variable names are used, or $[]A$ if de Bruijn indices are used). The λ-calculus à la de Bruijn is only given using variable names, for the version using de Bruijn indices see [23].

2.1 Rewriting notions

All the systems of this paper have a common feature. First, the syntax (the set of terms, types, substitutions, etc.) is given and then a set of rules that work on the syntax is presented. Those rules are *rewrite rules* and are of the form $A \to_R B$ or $(A, B) \in R$ if we prefer to talk of rewrite relations. These rules take a certain expression of the syntax (term, type, substitution, etc.) that matches the pattern of the left-hand side A of the rule and rewrite it in a way that matches the right-hand side B of the rule. This rewriting must take place inside larger formulas as well. For example, assume that A rewrites to B, then we must also be able to rewrite AC to BC. For this reason, an important notion for rewriting relations is that of *compatibility*. We introduce this notion here for the classical λ-calculus whose only operators are application and abstraction (the syntax is given in Definitions 2.9 and 2.39):

DEFINITION 2.1 (Compatibility for the classical λ-calculus)
We say that a binary relation R on the classical λ-calculus is compatible iff for all terms A, B of the λ-calculus and variable v, the following holds:

$$\frac{(A,B) \in R}{(AC,BC) \in R} \qquad \frac{(A,B) \in R}{(CA,CB) \in R} \qquad \frac{(A,B) \in R}{(\lambda v.A, \lambda v.B) \in R}(1) \qquad \frac{(A,B) \in R}{(\lambda A, \lambda B) \in R}(2)$$

(1) is in the case of variable names and (2) is in the case of de Bruijn indices.

This notion of compatibility will be extended according to the extra operations that will be added to the λ-calculus. For example, if we add substitution, we will have to add an extra clause on the compatibility for the case of the substitution operator.

DEFINITION 2.2 (Reduction notation)
Let S be a set and R a binary relation on S. We call R a reduction notion on S and use the following notation and definitions:

1. $\to_R$ is the compatible closure of R, and $(S, \to_R)$ is a reduction system.
2. $\overset{\epsilon}{\to}_R$ or just $\overset{\epsilon}{\to}$, is the reflexive closure $\to_R$.
3. $\overset{+}{\twoheadrightarrow}_R$ or just $\overset{+}{\twoheadrightarrow}$ is the transitive closure of $\to_R$.
4. $\twoheadrightarrow_R$ or just $\twoheadrightarrow$ is the reflexive and transitive closure of $\to_R$. When $A \twoheadrightarrow B$ we say there exists a *reduction sequence* from A to B .
5. $=_R$ is the reflexive, symmetric and transitive closure of $\to_R$. That is, $=_R$ is the least equivalence relation containing $\to_R$.
6. $\equiv$ is syntactic identity, and $A \equiv B$ means A and B are syntactically identical.

7. we write $A \overset{n}{\twoheadrightarrow}_R B$ or just $A \overset{n}{\twoheadrightarrow} B$ when the reduction sequence consists of $n \geq 0$ steps of reduction. We call n the length of the reduction sequence. i.e. if $n \geq 2$, there exists $B_1, \ldots, B_{n-1}$ such that $A \to_R B_1 \to_R \cdots \to_R B_{n-1} \to_R B$. When $n = 1$, $A \overset{1}{\twoheadrightarrow}_R B$ means $A \to_R B$. When $n = 0$, $A \overset{0}{\twoheadrightarrow}_R B$ means $A \equiv B$.
8. When $(A, B) \in R$, we say that A is an R-redex. In many cases, we introduce R as a set of rewrite rules of the form $A \to_R B$.
9. $A \in S$ is an *R-normal form* (R-nf for short) if A does not contain any R-redex.
10. We say that B *is an R-normal form of A* or A *has the R-normal form B* 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 two definitions help us describe this phenomenon:

DEFINITION 2.3 (Diamond property)
Let R be a binary relation on S. We say that $(S, \to_R)$ satisfies the *diamond property* if for all $A, B, C \in S$, if $A \to_R B$ and $A \to_R C$, then there is a D such that $B \twoheadrightarrow_R D$ and $C \twoheadrightarrow_R D$. Pictorially this is as follows:

$$\begin{tikzcd}
& A \arrow[dl, "R"'] \arrow[dr, "R"] & \\
B \arrow[dr] & & C \arrow[dl] \\
& D &
\end{tikzcd}$$

DEFINITION 2.4 (Confluence and Church Rosser)
Let R be a notion of reduction on S. We define local confluence (or Weak Church Rosser WCR), confluence (or Church Rosser CR) and strong confluence (or Strong Church Rosser SCR) as follows:

1. WCR: $\to_R$ satisfies the diamond property, i.e.:
$\forall A, B, C \in S \ \exists D \in S \ : (A \to_R B \ \wedge \ A \to_R C) \Rightarrow (B \twoheadrightarrow_R D \ \wedge \ C \twoheadrightarrow_R D)$.
2. CR: $\twoheadrightarrow_R$ satisfies the diamond property, i.e.:
$\forall A, B, C \in S \ \exists D \in S \ : (A \twoheadrightarrow_R B \ \wedge \ A \twoheadrightarrow_R C) \Rightarrow (B \twoheadrightarrow_R D \ \wedge \ C \twoheadrightarrow_R D)$.
3. SCR: $\forall A, B, C \in S \ \exists D \in S \ : (A \to_R B \wedge A \to_R C) \Rightarrow (B \to_R D \wedge C \to_R D)$.

LEMMA 2.5
Let R be a notion of reduction. If $\to_R$ is SCR then $\twoheadrightarrow_R$ is also SCR.

THEOREM 2.6
Let R be a notion of reduction that is CR. The following holds:

- If $A =_R B$ then there is a C such that $A \twoheadrightarrow_R C$ and $B \twoheadrightarrow_R C$.[1]
- If $A =_R B$ and B is in R-normal form, then $A \twoheadrightarrow_R B$.
- If $A =_R B$ then either A and B do not have R-normal forms or A and B have the same R-normal form.

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

- If A has R-normal forms B and C, then B and C are identical up to variable renaming. Hence, we speak of the R-normal form of A and denote it $R(A)$.
- $A =_R B$, A and B are in R-normal forms then A and B are identical up to variable renaming.

A second very important concern of reduction (or rewrite) systems 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 $n \to n+1$ applied to 1 will not terminate. Termination is a crucial property for implementation purposes. If an expression does not always terminate in a particular reduction system, perhaps it can terminate with some careful order of rules. Those expressions that will never terminate are disastrous for computation. We set the way with the following definition.

DEFINITION 2.7 (Normalization)
Let R be a reduction notion on S. We say that:

- A term A is *strongly R-normalizing* if there are no infinite R-reduction sequences starting at A.
- R is *strongly normalizing* (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 strongly R-normalizes.
- R is *weakly normalizing* (WN) if every $A \in S$ has an R-normal form.

When no confusion arises, R may be omitted and we speak simply of normal forms or normalization.

Strong normalization implies weak normalization and therefore the existence of normal forms. The following lemma is an important connection between strong normalization and confluence (its proof can be found in [3], proposition 3.1.25):

LEMMA 2.8 (Newman)
Every strongly normalizing, locally confluent notion of reduction relation is confluent. In other words, SN + WCR $\Longrightarrow$ CR.

2.2 *Classical λ-calculus with variable names*

DEFINITION 2.9 (Syntax of λ-terms)
The set of classical λ-terms or λ-expressions $\mathcal{M}$ is given by: $\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$ where $\mathcal{V} = \{x, y, z, \ldots\}$ is an infinite set of *term variables*. We let $v, v', v'', \cdots$ range over $\mathcal{V}$ and $A, B, C \cdots$ range over $\mathcal{M}$.

EXAMPLE 2.10
$(\lambda x.x)$, $(\lambda x.(xx))$, $(\lambda x.(\lambda y.x))$, $(\lambda x.(\lambda y.(xy)))$, and $((\lambda x.x)(\lambda x.x))$ are all classical λ-expressions.

This simple language is surprisingly rich. Its richness comes from the freedom to create and apply functions, especially higher order functions to other functions (and even to themselves). To explain the intuitive meaning of these three sorts of expressions, let us imagine a model where every λ-expression denotes an element of that model (which is a function). In particular, the variables denote a function in the model via an interpretation function or an *environment* which maps every variable into a specific element of the model. Such a model, by the way, was not obvious for more than forty years. In fact, for a domain D to be a model of

λ-calculus, it requires that the set of functions from D to D be included in D. Moreover, as the λ-calculus represents precisely the recursive functions, we know from Cantor's theorem that the domain D is much smaller than the set of functions from D to D. Dana Scott was armed by this theorem in his attempt to show the non-existence of the models of the λ-calculus. To his surprise, he proved the opposite of what he set out to show. He found in 1969 a model which has opened the door to an extensive area of research in computer science. We will not go into the details of these models in this paper.

DEFINITION 2.11 (Meaning of terms)
Here is now the intuitive meaning of each of the three λ-expressions given in the syntax:

Variables Functions denoted by variables are determined by what the variables are bound to in the *environment*. Binding is done by λ-abstraction.

Function application If A and B are λ-expressions, then so is (AB). This expression denotes the result of applying the function denoted by A to the function denoted by B.

Abstraction If v is a variable and A is an expression which may or may not contain occurrences of v, then $\lambda v.A$ denotes the function that maps the input value B to the output value $A[v := B]$.

EXAMPLE 2.12
$(\lambda x.x)$ denotes the identity function. $(\lambda x.(\lambda y.x))$ denotes the function which takes two arguments and returns the first.

As parentheses are cumbersome, we will use the following notational convention:

DEFINITION 2.13 (Notational convention)
We use these notational conventions:

1. Functional application associates to the left. So ABC denotes $((AB)C)$.
2. The body of a λ is anything that comes after it. So, instead of $(\lambda v.(A_1 A_2 \dots A_n))$, we write $\lambda v.A_1 A_2 \dots A_n$.
3. A sequence of λs is compressed to one, so $\lambda xyz.t$ denotes $\lambda x.(\lambda y.(\lambda z.t))$.

As a consequence of these notational conventions we get:

1. Parentheses may be dropped: (AB) and $(\lambda v.A)$ are written AB and $\lambda v.A$.
2. Application has priority over abstraction: $\lambda x.yz$ means $\lambda x.(yz)$ and not $(\lambda x.y)z$.

2.2.1 Variables and substitution

We need to manipulate λ-expressions in order to get values. For example, we need to apply $(\lambda x.x)$ to y to obtain y. To do so, we use the β-rule which says that $(\lambda v.A)B$ evaluates to *the body* A where v is substituted by B, written $A[v := B]$. However, one has to be careful. Look at the following example:

EXAMPLE 2.14
Evaluating $(\lambda fx.fx)g$ to $\lambda x.gx$ is perfectly acceptable but evaluating $(\lambda fx.fx)x$ to $\lambda x.xx$ is not. ByDefinition 2.11, $\lambda fx.fx$ and $\lambda fy.fy$ have the same meaning and hence $(\lambda fx.fx)x$ and $(\lambda fy.fy)y$ must also have the same meaning. Moreover, their values must have the same meaning. However, if $(\lambda fx.fx)x$ evaluates to $\lambda x.xx$ and $(\lambda fy.fy)x$ evaluates to $\lambda y.xy$, then we easily see, according to Definition 2.11, that $\lambda x.xx$ and $\lambda y.xy$ have two different

meanings. The first takes a function and applies it to itself, the second takes a function y and applies x (whatever its value) to y.

We define the notions of *free* and *bound* variables which will play an important role in avoiding the problem above. In fact, the λ is a variable binder, just like $\forall$ in logic:

DEFINITION 2.15 (Free and bound variables)
For a λ-term C, the set of free variables $FV(C)$, and the set of bound variables $BV(C)$, are defined inductively as follows:

$$\begin{array}{lll@{\qquad\qquad}lll}
FV(v) & =_{def} & \{v\} & BV(v) & =_{def} & \emptyset \\
FV(\lambda v.A) & =_{def} & FV(A) - \{v\} & BV(\lambda v.A) & =_{def} & BV(A) \cup \{v\} \\
FV(AB) & =_{def} & FV(A) \cup FV(B) & BV(AB) & =_{def} & BV(A) \cup BV(B)
\end{array}$$

An occurrence of a variable v in a λ-expression is free if it is not within the scope of a $\lambda v.$[2], otherwise it is bound. For example, in $(\lambda x.yx)(\lambda y.xy)$, the first occurrence of y is free whereas the second is bound. Moreover, the first occurrence of x is bound whereas the second is free. In $\lambda y.x(\lambda x.yx)$ the first occurrence of x is free whereas the second is bound. A *closed term* is a λ-term in which all variables are bound.

Free and bound variables play an important role in the λ-calculus for many reasons:

1. Almost all λ-calculi identify terms that only differ in the name of their bound variables. For example, as $\lambda x.x$ and $\lambda y.y$ have according to Definition 2.11 the same meaning (the identity function), they are usually identified. We will see more on this when we will introduce α-conversion (cf. Definition 2.19).
2. Substitution has to be handled with care due to the distinct roles played by bound and free variables. After substitution, no free variable can become bound. For example, $(\lambda x.fx)[f := x]$ must not return $\lambda x.xx$, but $\lambda y.xy$. These two latter terms have different meanings. $\lambda y.xy$ is obtained by renaming the bound x in $\lambda x.fx$ to y, and then performing the substitution. Thus $(\lambda x.fx)[f := x]$ is the same as $(\lambda y.fy)[f := x]$ which in its turn is the same as $\lambda y.xy$.
3. There is no point in substituting for a bound variable. For example, what is the point of turning $(\lambda x.x)[x := y]$ into $\lambda y.y$? Or even more strange (and not allowed syntactically), replacing x by $\lambda y.y$ due to the substitution $(\lambda x.x)[x := \lambda y.y]$.

DEFINITION 2.16 (Substitution)
For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$\begin{array}{lcl}
v[v := B] & \equiv & B \\
v'[v := B] & \equiv & v \qquad\qquad \text{if } v \not\equiv v' \\
(AC)[v := B] & \equiv & A[v := B]C[v := B] \\
(\lambda v.A)[v := B] & \equiv & \lambda v.A \\
(\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \not\in FV(B) \text{ or } v \not\in FV(A)) \\
(\lambda v'.A)[v := B] & \equiv & \lambda v''.[v' := v''][v := A] \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \in FV(B) \text{ and } v \in FV(A)).
\end{array}$$

In the last clause, v'' is chosen to be the first variable $\not\in FV(AB)$. In the case when terms are identified modulo the names of their bound variables, then in the last clause of the above

[2]Notice that the v in λv is not an occurrence of v.

definition, any $v'' \notin FV(AB)$ can be taken. In implementation, however, this identification is useless and a particular choice of v'' has to be made.

EXAMPLE 2.17
Check that $(\lambda y.yx)[x := z] \equiv \lambda y.yz$, that $(\lambda y.yx)[x := y] \equiv \lambda z.zy$, and that $(\lambda y.yz)[x := \lambda z.z] \equiv \lambda y.yz$.

LEMMA 2.18 (Substitution for variable names)
Let $A, B, C \in \mathcal{M},\ x,\ y, \in \mathcal{V}$. For $x \neq y$ and $x \notin \mathtt{FV}(C)$, we have that: $A[x := B][y := C] \equiv A[y := C][x := B[y := C]]$.

2.2.2 Reduction

Three notions of reduction will be studied in this section. The first is α-reduction which identifies terms up to variable renaming. The second is β-reduction evaluates λ-terms. The third is η-reduction which is used to identify functions that return the same values for the same arguments (extensionality). β-reduction is used in every λ-calculus, whereas η-reduction and α-reduction may or may not be used.

DEFINITION 2.19 (Alpha reduction)
$\rightarrow_\alpha$ is defined to be the least compatible relation closed under the axiom:

$$(\alpha) \qquad \lambda v.A \rightarrow_\alpha \lambda v'.A[v := v'] \qquad \text{where } v' \notin FV(A)$$

EXAMPLE 2.20
$\lambda x.x \rightarrow_\alpha \lambda y.y$ but it is not the case that $\lambda x.xy \rightarrow_\alpha \lambda y.yy$.
Moreover, $\lambda z.(\lambda x.x)x \twoheadrightarrow_\alpha \lambda z.(\lambda y.y)x$.

Recall that $\lambda x.x \not\equiv \lambda y.y$ even though they represent the same function. They are actually identical modulo α-conversion, i.e. $\lambda x.x =_\alpha \lambda y.y$.

DEFINITION 2.21 (Beta reduction)
$\rightarrow_\beta$ is defined to be the least compatible relation closed under the axiom:

$$(\beta) \qquad (\lambda v.A)B \rightarrow_\beta A[v := B]$$

EXAMPLE 2.22
Check that $(\lambda x.x)(\lambda z.z) \rightarrow_\beta \lambda z.z$, that $(\lambda y.(\lambda x.x)(\lambda z.z))xy \twoheadrightarrow_\beta y$, and that both $\lambda z.z$ and y are β-normal forms.

There follows a lemma about the interaction of β-reduction and substitution.

LEMMA 2.23
Let $A, B, C, D \in \mathcal{M}$.

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

PROOF. By induction on the structure of A for 1, on the derivation $A \rightarrow_\beta B$ for 2. ■

DEFINITION 2.24 (Eta reduction and η-normal forms)
$\rightarrow_\eta$ is defined to be the least compatible relation closed under the axiom:

$$(\eta) \qquad \lambda v.Av \rightarrow_\eta A \qquad \text{for } v \notin FV(A)$$

EXAMPLE 2.25
$\lambda x.(\lambda z.z)x \to_\eta \lambda z.z$ but it is not the case that $\lambda x.xx \to_\eta x$.
Moreover $\lambda y.(\lambda x.(\lambda z.z)x)y \twoheadrightarrow_\eta \lambda z.z$.

We use $\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$ when both η and β are used. η-conversion equates two terms that have the same behaviour as functions and implies extensionality.

LEMMA 2.26 (Extensionality)
For v not free in A or B, if $Av =_{\beta\eta} Bv$ then $A =_{\beta\eta} B$.

PROOF. Let $Av =_{\beta\eta} Bv$. By compatibility, $\lambda v.Av =_{\beta\eta} \lambda v.Bv$. Hence $A =_{\beta\eta} B$ by η. ∎

2.2.3 Meta theory

Example 2.27 below shows that not all expressions have normal forms (1), one may reduce terms using different reduction orders (2), the order of reduction will affect our reaching a normal form (3), and reducing a λ-expression may even result in a bigger expression rather than a smaller one (4). We underline the contracted redexes:

EXAMPLE 2.27

1. $\underline{(\lambda x.xx)(\lambda x.xx)}$ is not normalizing (and hence is not strongly normalizing). Hence, we know that this term does not have a normal form.
2. We can reduce in different orders:
   $(\lambda y.\underline{(\lambda x.x)(\lambda z.z)})xy \to_\beta \underline{(\lambda y.\lambda z.z)xy} \to_\beta \underline{(\lambda z.z)y} \to_\beta y$ and
   $\underline{(\lambda y.(\lambda x.x)(\lambda z.z))x}y \to_\beta (\underline{(\lambda x.x)(\lambda z.z)})y \to_\beta \underline{(\lambda z.z)y} \to_\beta y$
3. A term may be normalizing but not strongly normalizing:
   $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_\beta z$ yet
   $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta \ldots$
4. A term may grow after reduction:
$$\begin{array}{rcl} \underline{(\lambda x.xxx)(\lambda x.xxx)} & \to_\beta & \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx) \\ & \to_\beta & \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)(\lambda x.xxx) \\ & \to_\beta & \ldots \end{array}$$

Over expressions whose evaluation does not terminate, there is little we can do, so let us restrict our attention to those expressions whose evaluation terminates. β- and η-reduction can be seen as defining the steps that can be used for evaluating expressions to values. The values are intended to be themselves terms that cannot be reduced any further. Luckily, all orders lead to the same value (or normal form) of the expression for R-reduction where $R \in \{\beta, \beta\eta\}$. That is, if an expression R-reduces in two different ways to two values, then those values, if they are in R-normal form are the same (up to α-conversion).

EXAMPLE 2.28
Here are some ways to reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$. In all cases, the same final answer is obtained.

1. $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda x.x) \to_\beta$
   $\underline{(\lambda yz.z(yz))(\lambda x.x)} \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz$.
2. $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta$
   $\lambda z.\underline{(\lambda x.x)z}((\lambda x.x)z) \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz$.

3. $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta$
$\lambda z.(\lambda x.x)z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.\underline{(\lambda x.x)z}z \to_\beta \lambda z.zz.$

We would like that if A β- or $\beta\eta$-reduces to B and to C, then B and C β- or $\beta\eta$-reduce to the same term D. Luckily, the λ-calculus satisfies this property:

THEOREM 2.29 (β- and $\beta\eta$-reduction are Church Rosser)
For $R \in \{\beta, \beta\eta\}$, we have: $\forall A, B, C \in \mathcal{M}\ \exists D \in \mathcal{M}\ : (A \twoheadrightarrow_R B \wedge A \twoheadrightarrow_R C) \Rightarrow (B \twoheadrightarrow_R D \wedge C \twoheadrightarrow_R D)$.

PROOF. Various people have provided proofs of this theorem separately but Curry gave the first proof in [10]. [3] provides various proofs of this theorem. A shorter and new proof can be found in [46]. ∎

Due to this theorem (which says that the results of reductions do not depend on the order in which they are done), we may evaluate separate redexes in parallel.

THEOREM 2.30 (λ-calculus is consistent)
There are A, B such that $A \neq_{\beta(\eta)} B$.

PROOF. If $\lambda xy.x =_\beta \lambda xy.y$, then by Theorems 2.6 and 2.29, $\lambda xy.x =_\alpha \lambda xy.y$, but this is not the case. Hence $\lambda xy.x \neq_\beta \lambda xy.y$. ∎

So far we have answered two important questions.

1. Terms evaluate to unique values.
2. The λ-calculus is not trivial in the sense that it has more than one element.

Let us recall however from Example 2.27 that a term may have a normal form yet the evaluation order we use may not find this normal form. Hence the question now is: given a term that has a normal form, can we find this normal form? This is an important question because to be able to compute with the λ-caluclus, we must be able to find the normal form of a term if it exists. Luckily we have a positive result to this question. That is, if a term has a normal form then there is a reduction strategy that finds this normal form. The positive result is given by the normalization theorem (Theorem 2.36) which tells us that blind alleys in a reduction can be avoided by reducing the *leftmost* β or η-redex. That is, by reducing the redex whose beginning λ is as far to the left as possible. First, we need two sorts of redexes:

DEFINITION 2.31 (Left-most outermost redex)
The leftmost outermost redex of a term is the redex whose λ is the leftmost λ of the term. More precisely:

- $lmo(AB) =_{def} AB$ if AB is a β-redex;
- $lmo(AB) =_{def} lmo(A)$ if AB is not a β-redex;
- $lmo(\lambda v.A) =_{def} lmo(A)$.

DEFINITION 2.32 (Right-most innermost redex)
The rightmost innermost redex of a term is the redex whose λ is the rightmost λ of the term. More precisely:

- $rmi(AB) =_{def} rmi(B)$ if $rmi(B)$ is defined;
- $rmi(AB) =_{def} AB$ if $rmi(B)$ is not defined;

- $rmi(\lambda v.A) =_{def} rmi(A)$.

EXAMPLE 2.33

1. The leftmost outermost redex of $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ is the whole term itself and not $((\lambda x.xx)(\lambda x.xx))$.
2. The rightmost innermost redex of $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ is $((\lambda x.xx)(\lambda x.xx))$.
3. Inner and outer refer to the nesting of expressions. For example, the entire expression is the outermost redex in $(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)$ whereas the innermost redex is the subterm $(\lambda x.x)z$.

DEFINITION 2.34 (Normal-order reduction)
A reduction sequence is *normal order* or *call by name* if the leftmost redex is always reduced.

DEFINITION 2.35 (Applicative-order reduction)
A reduction sequence is *applicative order* or *call by value* if the rightmost redex is always reduced.

According to the call by value strategy, an argument is called only if it is a value (a normal form). According to the call by name strategy, an argument is called without first computing its value. Normal order reduction is guaranteed to reach a normal form if it exists. Applicative order however, might get stuck forever evaluating a term that is not strongly normalizing (but may be normalizing). For example, if normal order is used, $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ will yield z; it will never terminate on the other hand, if applicative order is used. Applicative order however can reach a normal form faster than normal order. For example, take $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$.

1. Applicative: $(\lambda x.xx)((\lambda y.y)(\lambda z.z))\to_\beta(\lambda x.xx)(\lambda z.z)\to_\beta(\lambda z.z)(\lambda z.z)\to_\beta\lambda z.z$.
2. Normal: $(\lambda x.xx)((\lambda y.y)(\lambda z.z))\to_\beta((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z))\to_\beta$ $(\lambda z.z)((\lambda y.y)(\lambda z.z))\to_\beta(\lambda y.y)(\lambda z.z)\to_\beta\lambda z.z$.

The normalization theorem (cf. [3]) states that if a term has a normal form then it is found by the leftmost outermost reduction strategy (which is not the most efficient):

THEOREM 2.36 (Normalization theorem)
If A has a normal form, then iterated contraction of the leftmost redex leads to that normal form.

2.3 Classical λ-calculus with de Bruijn indices

As we have seen in the previous section, 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 three methods:

- 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. We will not touch combinators in this paper. The interested reader can refer to [21].

- The use of the Barendregt Variable Convention (VC) which makes it possible to rewrite substitution in a way which does not deal with renaming variables. (VC) assumes that if at some place we discuss the terms $A_1, A_2, \ldots, A_n$, then all the bound variables in these terms are different from the free ones and that $\lambda v \ldots \lambda v$ is never used; rather, one uses $\lambda v \ldots \lambda v'$. For example, instead of writing $(\lambda x.\lambda y.xy)x$, we write $(\lambda z.\lambda y.zy)x$.
  Due to (VC), the two clauses of Definition 2.16 get replaced by the single clause:
  $$(\lambda v'.A)[v := B] \equiv \lambda v'.A[v := B] \qquad \text{if } v' \not\equiv v$$
  (VC) hides the problem rather than solving it. All the calculations and variable renaming have to be done. But (VC) assumes there is some magical stick which does all this work. Of course, we cannot use such an assumption when we do real work with our terms especially when we are implementing them.
- The use of de Bruijn indices, which avoid clashes of variable names and therefore neither α-conversion nor Barendregt's convention, are needed. This is explained in detail in this section.

2.3.1 Syntax

De Bruijn noted that due to the fact that terms such as $\lambda x.x$ and $\lambda y.y$ 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 variable indices of the λs and to replace their occurrences in the body of the term by the number which represents how many λs one has to cross before one reaches the λ binding the particular occurrence at hand.

EXAMPLE 2.37

1. $\lambda x.x$ is replaced by $\lambda 1$. That is, x is removed, and the x of the body x is replaced by 1 to indicate the λ it refers to.
2. $\lambda x.\lambda y.xy$ is replaced by $\lambda\lambda 21$. That is, the x and y of λx and λy are removed whereas the x and y of the body xy are replaced by 2 and 1 respectively in order to refer back to the λs that bind them.
3. Similarly, $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.xz)$ is replaced by $\lambda(\lambda 1(\lambda 1))(\lambda 12)$.

Note that the above terms are all closed. What do we do if we had a term that has free variables? For example, how do we write $\lambda x.xz$ using de Bruijn's indices?

In the presence of free variables, a *free variable list* which orders the variables must be assumed. For example, assume we take $x, y, z, \ldots$ 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 λs in whose scope z is, and then we continue counting in the free variable list using the order assumed. The following example demonstrates:

EXAMPLE 2.38
$\lambda x.xz$, $(\lambda x.xz)y$ and $(\lambda x.xz)x$ translate respectively into $\lambda 14$, $(\lambda 14)2$ and $(\lambda 14)1$.

Now we are ready to define the classical λ-calculus with de Bruijn indices.

DEFINITION 2.39
We define Λ, the *set of terms with de Bruijn indices*, as follows:

$$\Lambda ::= \mathbb{N} \mid (\Lambda\Lambda) \mid (\lambda\Lambda) \qquad .$$

As for $\mathcal{M}$, we use $A, B, \ldots$ to range over Λ. We also use $m, n, \ldots$ to range over $\mathbb{N}$ (positive natural numbers). Conventions 1 and 2 of Definition 2.13 are used (without the dots of course) and the consequences of that definition also hold here.

2.3.2 Updating and substitution

In the classical λ-calculus with de Bruijn indices, variables are represented by de Bruijn indices (natural numbers). 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 among 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 2.40
Writing $\underline{(\lambda x\lambda y.zxy)(\lambda x.yx)} \to_\beta \lambda u.z(\lambda x.yx)u$ using de Bruijn indices, one gets $\underline{(\lambda\lambda\mathtt{521})(\lambda\mathtt{31})} \to_\beta \lambda\mathtt{4}(\lambda\mathtt{41})\mathtt{1}$. The body of $\lambda\lambda 521$ is $\lambda 521$ and the variable bound by the first λ of $\lambda\lambda 521$ is the 2. Hence, we need to replace in $\lambda 521$ the 2 by $\lambda 31$. But if we simply replace $\mathtt{2}$ in $\lambda\mathtt{521}$ by $\lambda\mathtt{31}$ we get $\lambda\mathtt{5}(\lambda\mathtt{31})\mathtt{1}$, which is not correct. We needed to decrease $\mathtt{5}$ as one λ disappeared and to increment the free variables of $\lambda\mathtt{31}$ as they occur within the scope of one more λ.

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

- find in A the occurrences $n_1, \ldots n_k$ of the variable bound by the λ of λA;
- decrease the variables of A to reflect the disappearance of the λ from λA;
- replace the occurrences $n_1, \ldots n_k$ in A by updated versions of B which take into account that 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 $(\lambda A)B \to_\beta A\{\!\{\mathtt{1} \leftarrow B\}\!\}$ and let us define $A\{\!\{\mathtt{1} \leftarrow B\}\!\}$ in a way that all the work of $1 \ldots 3$ above is carried out. We need counters described informally as follows:

1. We start traversing A (here $\lambda 521$) with a unique counter initialized at 1.
2. In arriving at an application node, we create a copy of the counter in order to have one counter for each branch.
3. In arriving at an abstraction node, we increment the counter.
4. In arriving at a leaf (i.e. a number):
   (a) If it is superior to the counter, we decrease it by 1, because there will be a 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 $(n-1)\lambda$s. We must therefore adjust the numbers of B so that we can modify the binding relations inside B. For this we use another family of functions that we call *meta-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 meta-updating functions.

DEFINITION 2.41
The *meta-updating functions* $U_k^i : \Lambda \to \Lambda$ for $k \geq 0$ and $i \geq 1$ are defined inductively as follows:

$$U_k^i(AB) \equiv U_k^i(A)\, U_k^i(B) \qquad U_k^i(\mathtt{n}) \equiv \begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \leq k\,. \end{cases}$$
$$U_k^i(\lambda A) \equiv \lambda(U_{k+1}^i(A))$$

The intuition behind U_k^i 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 meta-substitution functions:

DEFINITION 2.42
The *meta-substitutions at level* i, for $i \geq 1$, of a term $B \in \Lambda$ in a term $A \in \Lambda$, denoted $A\{\!\{\mathtt{i} \leftarrow B\}\!\}$, is defined inductively on a as follows:

$$\begin{array}{ll} (A_1A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} \equiv (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) \\ (\lambda A)\{\!\{\mathtt{i} \leftarrow B\}\!\} \quad \equiv \lambda(A\{\!\{\mathtt{i+1} \leftarrow B\}\!\}) \end{array} \qquad \mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} \equiv \begin{cases} \mathtt{n-1} & \text{if } n > i \\ U_0^i(B) & \text{if } n = i \\ \mathtt{n} & \text{if } n < i\,. \end{cases}$$

The first two equalities propagate the substitution through applications and abstractions and the last 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. It is easy to check for example that $(\lambda\mathtt{521})\{\!\{\mathtt{1} \leftarrow (\lambda\mathtt{31})\}\!\} \equiv \lambda\mathtt{4}(\lambda\mathtt{41})\mathtt{1}$ and hence $(\lambda\lambda\mathtt{521})(\lambda\mathtt{31}) \to_\beta \lambda\mathtt{4}(\lambda\mathtt{41})\mathtt{1}$.

The following lemma establishes the properties of the meta-substitutions and meta-updating functions. The proof of this lemma is obtained by induction on a and can be found in [26] (the proof of 3 requires 2 with $p = 0$; the proof of 4 uses 1 and 3 both with $k = 0$; finally, 5 with $p = 0$ is needed to prove 6).

LEMMA 2.43
1. For $k < n \leq k + i$ we have: $U_k^i(A) \equiv U_k^{i+1}(A)\{\!\{\mathtt{n} \leftarrow B\}\!\}$.
2. For $p \leq k < j + p$ we have: $U_k^i(U_p^j(A)) \equiv U_p^{j+i-1}(A)$.
3. For $i \leq n - k$ we have: $U_k^i(A)\{\!\{\mathtt{n} \leftarrow B\}\!\} \equiv U_k^i(A\{\!\{\mathtt{n-i+1} \leftarrow B\}\!\})$.
4. [Meta-substitution lemma] For $1 \leq i \leq n$ we have:
   $A\{\!\{\mathtt{i} \leftarrow B\}\!\}\{\!\{\mathtt{n} \leftarrow C\}\!\} \equiv A\{\!\{\mathtt{n+1} \leftarrow C\}\!\}\{\!\{\mathtt{i} \leftarrow B\{\!\{\mathtt{n-i+1} \leftarrow C\}\!\}\}\!\}$.
5. For $m \leq k + 1$ we have: $U_{k+p}^i(U_p^m(A)) \equiv U_p^m(U_{k+p+1-m}^i(A))$.
6. [Distribution lemma]
   For $n \leq k + 1$ we have: $U_k^i(A\{\!\{\mathtt{n} \leftarrow B\}\!\}) \equiv U_{k+1}^i(A)\{\!\{\mathtt{n} \leftarrow U_{k-n+1}^i(B)\}\!\}$.

Case 4 is the version of Lemma 2.18 using de Bruijn indices.

2.3.3 Reduction

DEFINITION 2.44
β-reduction is the least compatible relation on Λ generated by:

$$(\beta\text{-rule}) \qquad (\lambda A)\,B \to_\beta A\{\!\{\mathtt{1} \leftarrow B\}\!\}$$

The classical *λ-calculus with de Bruijn indices,* is the reduction system generated by the only rewriting rule β.

We say that the λ-calculi with variable names and with de Bruijn indices are isomorphic is there are translation functions between $\mathcal{M}$ and Λ which are inverses of each other and which preserve β-reductions. The following theorem establishes the isomorphism of the λ-calculi with variable names and de Bruijn indices (cf. [35] for a proof). We will discuss a similar isomorphism in Section 3.2.

THEOREM 2.45
The classical λ-calculus with de Bruijn indices and the classical λ-calculus with variable names are isomorphic.

THEOREM 2.46
The classical λ-calculus with de Bruijn indices is confluent.

PROOF. By the isomorphism stated in Theorem 2.45, the confluence of the classical λ-calculus with variable names (cf. [3] Theorem 3.2.8) is transportable to the classical λ-calculus à la de Bruijn. ∎

Finally, here is the version of Lemma 2.23 for de Bruijn indices. Note that we need not only to ensure the good passage of the β-rule through the meta-substitutions but also through the U_k^i.

LEMMA 2.47
Let $A, B, C, D \in \Lambda$.

1. If $C \to_\beta D$ then $i)\, U_k^i(C) \to_\beta U_k^i(D)$ 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 2.43.6. Case (ii) is by induction on A using (i). 2. Is by induction on A using Lemma 2.43.4. ∎

2.4 *From the classical λ-calculus with de Bruijn indices to a substitution calculus*

Having seen in Section 2.3 the meta-updating and meta-substitution operators, an approach to introduce explicit substitution to the λ-calculus with de Bruijn indices is to extend the syntax of Definition 2.39 to include new operators that internalize updating and substitution. This is done as follows:

DEFINITION 2.48 (Syntax of the λs-calculus)
Terms of the λs-calculus are given by:

$$\Lambda s ::= I\!N \mid (\Lambda s \Lambda s) \mid (\lambda \Lambda s) \mid (\Lambda s\, \sigma^i \Lambda s) \mid (\varphi_k^i \Lambda s) \qquad where \quad i \geq 1\,,\ \ k \geq 0\,.$$

We use the notational conventions defined earlier to get rid of unnecessary parenthesis.

Now, we need to include reduction rules that operate on the new terms built with updating and substitutions. Definitions 2.41 and 2.42 suggest these rules. The resulting calculus is the explicit substitution calculus λs of [26] whose set of rules is given in Figure 1. Note that these rules are nothing more than β written now as σ-generation, together with the rules of Definitions 2.41 and 2.42 oriented as expected.

DEFINITION 2.49
The set of rules λs is given in Figure 1. The *λs-calculus* is the reduction system $(\Lambda s, \to_{\lambda s})$ where $\to_{\lambda s}$ is the least compatible reduction on Λs generated by the set of rules λs.

σ-generation	$(\lambda A)\,B$	$\longrightarrow$	$A\,\sigma^1\,B$
σ-λ-transition	$(\lambda A)\,\sigma^i B$	$\longrightarrow$	$\lambda(A\,\sigma^{i+1}\,B)$
σ-app-transition	$(A_1\,A_2)\,\sigma^i B$	$\longrightarrow$	$(A_1\,\sigma^i B)\,(A_2\,\sigma^i B)$
σ-destruction	$\mathtt{n}\,\sigma^i B$	$\longrightarrow$	$\begin{cases} \mathtt{n-1} & \text{if } n > i \\ \varphi^i_0\,B & \text{if } n = i \\ \mathtt{n} & \text{if } n < i \end{cases}$
φ-λ-transition	$\varphi^i_k(\lambda A)$	$\longrightarrow$	$\lambda(\varphi^i_{k+1}\,A)$
φ-app-transition	$\varphi^i_k(A_1\,A_2)$	$\longrightarrow$	$(\varphi^i_k\,A_1)\,(\varphi^i_k\,A_2)$
φ-destruction	$\varphi^i_k\,\mathtt{n}$	$\longrightarrow$	$\begin{cases} \mathtt{n+i-1} & \text{if } n > k \\ \mathtt{n} & \text{if } n \le k \end{cases}$

FIGURE 1. The λs-rules

σ-σ-transition	$(A\,\sigma^i B)\,\sigma^j\,C$	$\longrightarrow$	$(A\,\sigma^{j+1}\,C)\;\sigma^i\,(B\,\sigma^{j-i+1}\,C)$	if	$i \le j$
σ-φ-transition 1	$(\varphi^i_k\,A)\,\sigma^j\,B$	$\longrightarrow$	$\varphi^{i-1}_k\,A$	if	$k < j < k+i$
σ-φ-transition 2	$(\varphi^i_k\,A)\,\sigma^j\,B$	$\longrightarrow$	$\varphi^i_k(A\,\sigma^{j-i+1}\,B)$	if	$k+i \le j$
φ-σ-transition	$\varphi^i_k(A\,\sigma^j\,B)$	$\longrightarrow$	$(\varphi^i_{k+1}\,A)\,\sigma^j\,(\varphi^i_{k+1-j}\,B)$	if	$j \le k+1$
φ-φ-transition 1	$\varphi^i_k\,(\varphi^j_l\,A)$	$\longrightarrow$	$\varphi^j_l\,(\varphi^i_{k+1-j}\,A)$	if	$l+j \le k$
φ-φ-transition 2	$\varphi^i_k\,(\varphi^j_l\,A)$	$\longrightarrow$	$\varphi^{j+i-1}_l\,A$	if	$l \le k < l+j$

FIGURE 2. The new rules of the λs_e-calculus

[26] establishes that the s-calculus (i.e. the reduction system whose rules are those of Figure 1 excluding σ-generation) is strongly normalizing, that the λs-calculus is confluent, simulates β-reduction and has the property of preservation of strong normalization PSN (i.e. if a term terminates in the calculus with de Bruijn indices presented in Section 2.3, then it terminates in the λs-calculus). If the λs-calculus is extended with open terms (variables that range over terms), then the reduction rules need also to be extended to guarantee confluence. This extension is essential for implementations, see [34, 36, 37]. Adding the six items of Lemma 2.43 as oriented rewriting rules results in the calculus λs_e which is confluent on open terms [28]. Like $\lambda\sigma$ of [1], this calculus does not satisfy PSN [18].

DEFINITION 2.50 (The λs_e-calculus)
Terms of the λs_e-calculus are given by:
$\Lambda s_{op} ::= \mathbf{V} | \mathbb{N} | (\Lambda s_{op} \Lambda s_{op}) | (\lambda \Lambda s_{op}) | (\Lambda s_{op}\,\sigma^j \Lambda s_{op}) | (\varphi^i_k \Lambda s_{op}) \quad where \quad j, i \ge 1\,, \quad k \ge 0$
and where **V** stands for a set of variables, over which X, Y, ... range.

The set of rules λs_e is obtained by adding the rules given in Figure 2 to the set λs of Figure 1. The *λs_e-calculus* is the reduction system $(\Lambda s_{op}, \rightarrow_{\lambda s_e})$ where $\rightarrow_{\lambda s_e}$ is the least compatible reduction on Λs_{op} generated by the set of rules λs_e.

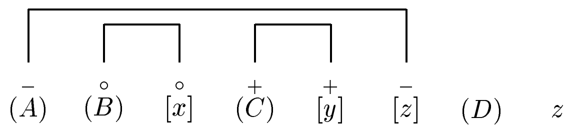

FIGURE 3. Redexes in de Bruijn's notation

2.5 The λ-calculus à la de Bruijn

De Bruijn departed from the classical notation of the λ-calculus seen so far. Instead, he wrote the argument before the function and often used $[x]$ instead of λx. Here is the translation from classical notation into de Bruijn's notation via $\mathcal{I}$.

$$\mathcal{I}(v) =_{def} v, \qquad \mathcal{I}(\lambda v.B) =_{def} [v]\mathcal{I}(B), \qquad \mathcal{I}(AB) =_{def} (\mathcal{I}(B))\mathcal{I}(A)$$

De Bruijn called items of the form (A) and $[v]$ *applicator wagon* respectively *abstractor wagon*, or simply *wagon*.

EXAMPLE 2.51
$\mathcal{I}((\lambda x.(\lambda y.xy))z) = (z)[x][y](y)x$. The wagons are (z), $[x]$, $[y]$ and (y).

In de Bruijn's notation, the β-rule $(\lambda v.A)B \to_\beta A[v := B]$ becomes:

$$(B)[v]A \to_\beta [v := B]A$$

Note that the applicator wagon (B) and the abstractor wagon $[v]$ occur NEXT to each other. Here is an example which compares β-reduction in both the classical and de Bruijn's notation. Wagons that have the same symbol on top, are matched.

Classical Notation	De Bruijn's Notation
$(\underline{\overset{\circ}{\lambda x} .(\overset{+}{\lambda y} . \overset{-}{\lambda z} .zD)\ \overset{+}{C})\ \overset{\circ}{B})}\ \overset{-}{A}$	$(\overset{-}{A})\ \underline{(\overset{\circ}{B})[\overset{\circ}{x}]}\ (\overset{+}{C})[\overset{+}{y}][\overset{-}{z}]\ (D)z$
$\downarrow_\beta$	$\downarrow_\beta$
$((\underline{\overset{+}{\lambda y} . \overset{-}{\lambda z} .zD)\ \overset{+}{C}})\ \overset{-}{A}$	$(\overset{-}{A})\ \underline{(\overset{+}{C})[\overset{+}{y}]}\ [\overset{-}{z}]\ (D)z$
$\downarrow_\beta$	$\downarrow_\beta$
$\underline{(\overset{-}{\lambda z} .zD)\ \overset{-}{A}}$	$\underline{(\overset{-}{A})[\overset{-}{z}]}(D)z$
$\downarrow_\beta$	$\downarrow_\beta$
AD	$(D)A$

The matching redexes in de Bruijn's notation are easily seen in the Figure 3.

The *bracketing structure* in classical notation of $((\overset{\circ}{\lambda x} .(\overset{+}{\lambda y} . \overset{-}{\lambda z} .zD)\ \overset{+}{C})\ \overset{\circ}{B})\ \overset{-}{A})$, is $\overset{\circ}{\{_1}\ \overset{+}{\{_2}\ \overset{-}{\{_3}\ \overset{+}{\}_2}\ \overset{\circ}{\}_1}\ \overset{-}{\}_3}$, where $\{_i$ and $\}_i$ match. Whereas $(\overset{-}{A})(\overset{\circ}{B})[\overset{\circ}{x}](\overset{+}{C})[\overset{+}{y}][\overset{-}{z}]\ (D)z$ has the simpler bracketing structure $\overset{-}{\{}\overset{\circ}{\{}\overset{\circ}{\}}\overset{+}{\{}\overset{+}{\}}\overset{-}{\}}$ in de Bruijn's notation. An applicator (A) and an abstractor $[v]$ are *partners* when they match like $\{$ and $\}$. Non-partnered wagons are *bachelors*. A sequence of wagons is called a *segment*. A segment is *well balanced* when it contains only partnered wagons.

EXAMPLE 2.52
Consider the term $\overset{-}{(A)}\overset{\circ}{(B)}\overset{\circ}{[x]}\overset{+}{(C)}\overset{+}{[y]}\overset{-}{[z]}\ (D)z$. The wagons (A), (B), $[x]$, (C), $[y]$, and $[z]$ are partnered and the wagon (D) is a bachelor. The segment $\overset{-}{(A)}\overset{\circ}{(B)}\overset{\circ}{[x]}\overset{+}{(C)}\overset{+}{[y]}\overset{-}{[z]}$ is well balanced.

The λ-calculus à la de Bruijn has many advantages over the classical λ-calculus. Some of these advantages are summarized in [25]. In what follows we mention some.

A. Structure of terms Each non-empty segment $\overline{s}$ has a unique *partitioning* into sub-segments $\overline{s} = \overline{s_0 s_1} \cdots \overline{s_n}$ such that

- For even i, the segment $\overline{s_i}$ is well balanced. For odd i, the segment $\overline{s_i}$ is a bachelor segment, i.e. it contains only bachelor main items.
- All well balanced segments after $\overline{s_0}$ and all bachelor segments are non-empty.
- If $\overline{s_i} = [v_1] \cdots [v_m]$ (only abstractor wagons) and $\overline{s_j} = (a_1) \cdots (a_p)$ (only applicator wagons), then $i < j$, i.e. $\overline{s_i}$ precedes $\overline{s_j}$ in $\overline{s}$.

EXAMPLE 2.53
$\overline{s} \equiv [x][y](A)[z][x'](B)(C)(D)[y'][z'](E)$, has the following partitioning:
well-balanced segment $\overline{s_0} \equiv \emptyset$, bachelor segment $\overline{s_1} \equiv [x][y]$, well-balanced segment $\overline{s_2} \equiv (A)[z]$, bachelor segment $\overline{s_3} \equiv [x'](B)$, well-balanced segment $\overline{s_4} \equiv (C)(D)[y'][z']$, bachelor segment $\overline{s_5} \equiv (E)$.

B. Generalized Reduction Looking at Figure 3, one sees that either (A) can be moved to the right to occur next to its partner $[z]$ or $[z]$ can be moved to the left to appear next to (A). One can instead generalise β-reduction so that the (extended) redex based on (A) and $[z]$ is fired before the other redexes. All these steps happen via rules like those listed in Figure 4. These rules have been studied by many researchers [24, 2, 17, 29, 30, 31, 44, 40, 48, 45, 50, 38]. De Bruijn's notation makes it clearer to describe generalized reduction as Figure 4 illustrates, where we assume Barendregt's variable conventions (see page 377).

C. Properties are easier to state in de Bruijn's notation We illustrate this point with the example of describing the s_e-normal forms where the s_e-calculus is the reduction system $(\Lambda s_{op}, \rightarrow_{s_e})$ where $\rightarrow_{s_e}$ is the least compatible reduction on Λs_{op} generated by the set of rules of Figures 1 and 2 excluding σ-generation. Theorem 2.54 gives the s_e-normal forms in classical notation. Theorem 2.55 gives them in de Bruijn's notation. These theorems are taken from [28].

THEOREM 2.54
A term $A \in \Lambda s_{op}$ is an s_e-normal form iff one of the following holds:

- $A \in \mathbf{V} \cup I\!N$, i.e. A is a variable or a de Bruijn number.
- $A \equiv B\,C$, where B and C are s_e-normal forms.
- $A \equiv \lambda B$, where B is an s_e-normal form.
- $A \equiv B\,\sigma^j C$, where C is an s_e-nf and B is an s_e-nf of the form X, or $D\,\sigma^i E$ with $j < i$, or $\varphi^i_k d$ with $j \leq k$.
- $A \equiv \varphi^i_k B$, where B is an s_e-nf of the form X, or $C\,\sigma^j d$ with $j > k+1$, or $\varphi^j_l C$ with $k < l$.

PROOF. By analysing the structure of A. ∎

Name	In classical notation	In de Bruijn's notation
(θ)	$((\overset{-}{\lambda} x\ .\ \overset{+}{C})\ \overset{-}{B})\ \overset{+}{A}$ $\downarrow$ $(\overset{-}{\lambda} x\ .\ \overset{+}{C}\overset{+}{A})\ \overset{-}{B}$	$(\overset{+}{A})(\overset{-}{B})[\overset{-}{x}]\overset{+}{C}$ $\downarrow$ $(\overset{-}{B})[\overset{-}{x}](\overset{+}{A})\overset{+}{C}$
(γ)	$(\overset{+}{\lambda} x\ .\lambda y.C)\ \overset{+}{B}$ $\downarrow$ $\lambda y.(\overset{+}{\lambda} x\ .C)\ \overset{+}{B}$	$(\overset{+}{B})[\overset{+}{x}]\ [y]C$ $\downarrow$ $[y]\ (\overset{+}{B})[\overset{+}{x}]\ C$
(g)	$((\overset{-}{\lambda} x\ .\ \overset{+}{\lambda} y\ .C)\ \overset{-}{B})\ \overset{+}{A}$ $\downarrow$ $(\overset{-}{\lambda} x\ .C[y := A])\ \overset{-}{B}$	$(\overset{+}{A})(\overset{-}{B})[\overset{-}{x}][\overset{+}{y}]\ C$ $\downarrow$ $(\overset{-}{B})[\overset{-}{x}]\ [y := A]C$
(γ_C)	$((\overset{+}{\lambda} x\ .\ \overset{-}{\lambda} y\ .C)\ \overset{+}{B})\ \overset{-}{A}$ $\downarrow$ $(\overset{-}{\lambda} y\ .(\overset{+}{\lambda} x\ .C)\ \overset{+}{B})\ \overset{-}{A}$	$(\overset{-}{A})(\overset{+}{B})[\overset{+}{x}][\overset{-}{y}]\ C$ $\downarrow$ $(\overset{-}{A})[\overset{-}{y}](\overset{+}{B})[\overset{+}{x}]\ C$

FIGURE 4. Generalized reduction

There is a simple way to describe the s_e-nf's using de Bruijn's notation. First, note that in de Bruijn's notation $A\,\sigma^i B$ and $\varphi^i_k A$ are written respectively as: $(B\,\sigma^i)A$ and $(\varphi^i_k)A$. The parts $(B\,\sigma^i)$ and (φ^i_k) are called σ- and φ-wagons respectively. The subterms B and A are the *bodies* of these respective wagons.

A *normal $\sigma\varphi$-segment* $\overline{s}$ is a sequence of σ- and φ-wagons such that every pair of adjacent wagons in $\overline{s}$ is of the form:

$(\varphi^i_k)(\varphi^j_l)$ and $k < l$	$(\varphi^i_k)(B\,\sigma^j)$ and $k < j - 1$
$(B\,\sigma^i)(C\,\sigma^j)$ and $i < j$	$(B\,\sigma^j)(\varphi^i_k)$ and $j \leq k$

e.g. $(\varphi^2_3)(\varphi^1_4)(\varphi^6_7)(B\sigma^9)(C\sigma^{11})(\varphi^2_{11})(\varphi^5_{16})$ and $(B\sigma^1)(C\sigma^3)(D\sigma^4)(\varphi^2_5)(\varphi^1_6)(\varphi^4_7)(A\sigma^{10})$ are normal $\sigma\varphi$-segments.

THEOREM 2.55
The s_e-nfs can be described by the following syntax:

$$NF ::= \mathbf{V} \mid I\!\!N \mid (NF)NF \mid [\,]NF \mid \overline{s}\,\mathbf{V}$$

where $\overline{s}$ is a normal $\sigma\varphi$-segment whose bodies belong to NF.

3 Pure type systems

We have seen so far the type-free λ-calculus. Types however, aid in writing correct and terminating programs. Another influential role that types play is in their identifications with propositions in the paradigm of *propositions-as-types* due to Curry, Howard and de Bruijn. Under this paradigm, the problem of proof checking can be reduced to the problem of type checking in a programming language.

There are 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 $\lambda x.x$. 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. In this paper, we consider typing à la Church. We present what is known as *Pure Type Systems* or PTSs. Important type systems that are PTSs include Church's simply typed λ-calculus [8] and the calculus of constructions [9] which are also systems of the Barendregt cube [4]. Berardi [5] and Terlouw [47] have independently generalized the method of generating type systems into the pure type systems framework. This generalization has many advantages. First, it enables one to introduce eight logical systems that are in close correspondence with the systems of the Barendregt cube. Those eight logical systems can each be described as a PTS in such a way that the propositions-as-types interpretation obtains a canonical system form [4]. Second, the general setting of the PTSs makes it easier to write various proofs about the systems of the cube.

In the following sections of the present paper we will briefly review the classical PTS with variable names [4] and those with de Bruijn indices [27], essentially to state their isomorphism. This is a result of [27] to which we refer for all omitted proofs.

3.1 Classical pure type systems with variable names

All this section is taken from [4] where all the proofs can be found.

DEFINITION 3.1
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 $\{x, y, z, \ldots\}$ and $\mathcal{C}$ a set of constants over which, $c, c_1, \ldots$ range. We use $A, B, \ldots$ to range over $\mathcal{T}$ and $v, v', v'', \ldots$ to range over $\mathcal{V}$. Throughout, we take $\pi \in \{\lambda, \Pi\}$.

DEFINITION 3.2 (Free and bound variables)
Free and bound variables in terms are defined similarly to those of Definition 2.15 with the exception that $FV(c) =_{def} BV(c) =_{def} \emptyset$ and in the case of abstraction, $FV(\pi v : A.B) =_{def} (FV(B) \setminus \{v\}) \cup FV(A)$ and $BV(\pi v : A.B) =_{def} BV(A) \cup BV(B) \cup \{v\}$.

We write $A[x := B]$ to denote the term where all the free occurrences of x in A have been replaced by B. Furthermore, we take terms to be equivalent up to variable renaming. We assume moreover, the Barendregt variable convention (already discussed on page 377) which is formally stated as follows:

CONVENTION 3.3
(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 λs have different variables as subscript. Hence, we will not have $(\lambda x : A.x)x$, but $(\lambda y : A.y)x$ instead.

The definition of compatibility of a reduction relation for PTSs is that of the type-free calculus (given in Definition 2.1) but where the case of abstraction is replaced by:

$$\frac{(A_1, A_2) \in R}{(\pi x : A_1.B, \pi x : A_2.B) \in R} \qquad\qquad \frac{(B_1, B_2) \in R}{(\pi x : A.B_1, \pi x : A.B_2) \in R}$$

DEFINITION 3.4
β-reduction is the least compatible relation on $\mathcal{T}$ generated by

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

Now, we define some machinery needed for typing.

DEFINITION 3.5
1. A *statement* is of the form $A : B$ with $A, B \in \mathcal{T}$. We call A is the *subject* and B is 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.
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 3.6
A *type assignment* relation is a relation between a pseudo-context and two 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 3.7
Let Γ be a pseudo-context, A be a pseudo-term 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.

DEFINITION 3.8
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 3.9
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 axiomatized by the axioms and rules of Figure 5.

Each of the eight systems of the cube is obtained by taking $\mathcal{S} = \{*, \Box\}$, $\mathcal{A} = \{*, \Box\}$, and R to be a set of rules of the form (s_1, s_2, s_2) for $s_1, s_2 \in \{*, \Box\}$. We denote rules of the form (s_1, s_2, s_2) by (s_1, s_2). This means that the only possible (s_1, s_2) rules in the set R (in the case of the cube) are elements of the following set: $\{(*, *), (*, \Box), (\Box, *), (\Box, \Box)\}$. The basic system is the one where $(s_1, s_2) = (*, *)$ is the only possible choice. All other systems have this version of the formation rules, plus one or more other combinations of $(*, \Box)$, $(\Box, *)$ and $(\Box, \Box)$ for (s_1, s_2). See Figures 6 and 7.

Now, we list some of the properties of PTSs with variable names (see [4] for proofs). In Section 3.2, we will establish these properties for PTSs with de Bruijn indices.

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

FIGURE 5. PTSs with variables names

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

FIGURE 6. Different type formation conditions

LEMMA 3.10
Let $A, B \in \mathcal{T}$. If $A \to_\beta B$ then $FV(B) \subseteq FV(A)$.

THEOREM 3.11 (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 3.12 (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. The $x_1, \ldots, x_n$ are all distinct.
2. $FV(B), FV(C) \subseteq \{x_1, \ldots, x_n\}$.

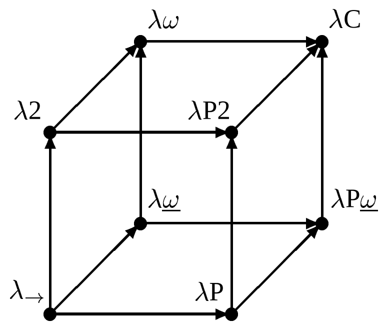

FIGURE 7. The cube

3. $FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $1 \le i \le n$.

THEOREM 3.13 (Subject Reduction SR, for PTSs with variable names)
If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$.

3.2 Classical pure type systems with de Bruijn indices

In this section, we will introduce pure type systems with de Bruijn indices and establish the isomorphism between them and those with variable names. All this section is taken from [27] where all the proofs can be found in detail.

DEFINITION 3.14
We define $\mathcal{T}$, the *set of pseudo-terms with de Bruijn indices*, by the syntax:

$$\mathcal{T} ::= I\!N \mid \mathcal{C} \mid (\mathcal{T}\,\mathcal{T}) \mid (\lambda\mathcal{T}.\mathcal{T}) \mid (\Pi\mathcal{T}.\mathcal{T}),$$

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 $I\!N$ (positive natural numbers). Again, we take $\pi \in \{\Pi, \lambda\}$.

DEFINITION 3.15
The *updating functions* $U^i_k : \mathcal{T} \to \mathcal{T}$ for $k \ge 0$ and $i \ge 1$ are defined as in Definition 2.41 for the case of the type-free λ-calculus, but with the addition of a clause for constants and the replacement of the abstraction rule as follows:

$$U^i_k(c) \equiv c \quad \text{for } c \in \mathcal{C} \qquad \text{and} \qquad U^i_k(\pi A.B) \equiv \pi U^i_k(A).(U^i_{k+1}(B))$$

DEFINITION 3.16
The *meta-substitutions at level* j, for $j \ge 1$, of a term $B \in \mathcal{T}$ in a term $A \in \mathcal{T}$, denoted $A\{\!\{\mathtt{j} \leftarrow B\}\!\}$, is defined inductively on A as in Definition 2.42 for the case of the type free λ-calculus, but with the addition of a clause for constants and the replacement of the abstraction rule as follows:

$$c\{\!\{\mathtt{j} \leftarrow B\}\!\} \equiv c \quad \text{for } c \in \mathcal{C} \quad \text{and} \quad (\pi A.C)\{\!\{\mathtt{j} \leftarrow B\}\!\} \equiv \pi A\{\!\{\mathtt{j} \leftarrow B\}\!\}).(C\{\!\{\mathtt{j+1} \leftarrow B\}\!\})$$

DEFINITION 3.17
β-reduction is the least compatible reduction on $\mathcal{T}$ generated by:

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

Note that we use $\rightarrow_\beta$ 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.

We now define the set of free variables of a term with de Bruijn indices. We write $N \setminus k$ to mean $\{n - k : n \in N, n > k\}$.

DEFINITION 3.18
The *set of free variables* of a term with de Bruijn indices is defined by induction as follows:

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

The following lemma on T corresponds to Lemma 3.10 on $\mathcal{T}$.

LEMMA 3.19
Let $A, B \in T$. If $A \rightarrow_\beta B$ then $FV(B) \subseteq FV(A)$.

Definition 3.5 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 3.6, 3.7 and 3.8 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 3.20
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 axiomatized by the axioms and rules of Figure 8.

Note 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 tradition of type systems in de Bruijn notation (cf. [1, 41]).

Note also the role played by the updating U_0^2 in the rules *(start), (weakening)*. This function increases by 1 the de Bruijn indices which correspond to free variables and its occurrence in these two rules is reasonable since the corresponding environments have been augmented by the addition of a new component. For example, $* \vdash 1 : *$. Hence, $1, * \vdash 1 : 2$. Moreover, $1, * \vdash 1 : 2$ and $* \vdash 1 : *$, hence $*, 1, * \vdash 2 = U_0^2(1) : 3 = U_0^2(2)$.

The following lemma (cf. [27]) is the equivalent, for de Bruijn indices, of Lemma 3.12.

LEMMA 3.21
Let $A_1, \ldots, A_n \vdash B : C$ then $FV(B), FV(C) \subseteq \{1, \ldots, n\}$ and, for $0 \leq i \leq n-1$, $FV(A_{n-i}) \subseteq \{1, \ldots, i\}$.

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 environment Γ written using variable names, into a term $t_1(A)$ and an environment $t(\Gamma)$ written with de Bruijn indices. We then prove that these translations preserve β-reduction (if in $\mathcal{T}$, $A \rightarrow_\beta B$ then in T, $t_1(A) \rightarrow_\beta t_1(B)$) and type assignment (if in $\mathcal{T}$, $\Gamma \vdash A : B$ then in T, $t(\Gamma) \vdash t_1(A) : t_1(B)$).
2. We define translations u_1 and u in the other sense and also prove preservation of β-reduction and type assignment.
3. We prove that these translations are inverses of each other.

In the rest of this paper, $[x_1, \ldots, x_n]$ stands for the ordered list of $x_1, \ldots, x_n$.

$$
\begin{array}{lll}
\text{(axioms)} & \vdash c : s & \text{if} \quad c : s \in \mathcal{A} \\[1em]
\text{(start)} & \dfrac{\Gamma \vdash A : s}{A, \Gamma \vdash \mathtt{1} : U_0^2(A)} & \\[1em]
\text{(weakening)} & \dfrac{\Gamma \vdash B : C \qquad \Gamma \vdash A : s}{A, \Gamma \vdash U_0^2(B) : U_0^2(C)} & \\[1em]
\text{(product)} & \dfrac{\Gamma \vdash A : s_1 \qquad A, \Gamma \vdash B : s_2}{\Gamma \vdash (\Pi A.B) : s_3} & \text{if} \quad (s_1, s_2, s_3) \in \mathcal{R} \\[1em]
\text{(application)} & \dfrac{\Gamma \vdash F : (\Pi A.B) \qquad \Gamma \vdash C : A}{\Gamma \vdash F\,C : B\{\!\{\mathtt{1} \leftarrow C\}\!\}} & \\[1em]
\text{(abstraction)} & \dfrac{A, \Gamma \vdash C : B \qquad \Gamma \vdash (\Pi A.B) : s}{\Gamma \vdash (\lambda A.C) : (\Pi A.B)} & \\[1em]
\text{(conversion)} & \dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'} &
\end{array}
$$

FIGURE 8. PTSs with de Bruijn indices

3.2.1 Translating $\mathcal{T}$ to T

DEFINITION 3.22 (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:

$$
\begin{aligned}
&t_{[x_1,\ldots,x_n]}(c) =_{def} c \quad \text{for} \quad c \in \mathcal{C} \\
&t_{[x_1,\ldots,x_n]}(v_i) =_{def} \min\{j \text{ such that } v_i = x_j\} \\
&t_{[x_1,\ldots,x_n]}(A\,B) =_{def} t_{[x_1,\ldots,x_n]}(A) t_{[x_1,\ldots,x_n]}(B) \\
&t_{[x_1,\ldots,x_n]}(\pi x : B.A) =_{def} \pi t_{[x_1,\ldots,x_n]}(B).t_{[x,x_1,\ldots,x_n]}(A) \text{ for } \pi \in \{\Pi, \lambda\}
\end{aligned}
$$

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

Note that Definition 3.22 is a good definition thanks to Lemma 3.12.

LEMMA 3.23
Let $A, B \in \mathcal{T}$ such that $FV(A) \subseteq \{x_1, \ldots x_n\}$ and $A \to_\beta B$.
Then $t_{[x_1,\ldots,x_n]}(A) \to_\beta t_{[x_1,\ldots,x_n]}(B)$.

THEOREM 3.24
Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ such that $\Gamma \vdash A : B$.
Then $t(\Gamma) \vdash t_{[x_n,\ldots,x_1]}(A) : t_{[x_n,\ldots,x_1]}(B)$.

3.2.2 Translating $\mathtt{T}$ to $\mathcal{T}$

DEFINITION 3.25 (The translation u)
Let $A \in \mathtt{T}$ 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:

$$u_{[x_n,\ldots,x_1]}(c) =_{def} c \quad \text{for } c \in \mathcal{C} \qquad\qquad u_{[x_n,\ldots,x_1]}(\mathtt{i}) =_{def} x_i$$
$$u_{[x_n,\ldots,x_1]}(A\,B) =_{def} u_{[x_n,\ldots,x_1]}(A)u_{[x_n,\ldots,x_1]}(B)$$
$$u_{[x_n,\ldots,x_1]}(\pi B.A) =_{def} \pi x : u_{[x_n,\ldots,x_1]}(B).u_{[x_n,\ldots,x_1,x]}(A) \quad \text{with } x \notin \{x_1, \ldots, x_n\}.$$

Note that Definition 3.25 is correct since $FV(\pi B.A) \subseteq \{1, \ldots, n\}$ implies $FV(A) \subseteq \{1, \ldots, n+1\}$. Furthermore, [27] proves that the definition for abstractions and products does not depend on the choice of the variable x.

DEFINITION 3.26
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 3.26 is correct thanks to Lemma 3.21.

LEMMA 3.27
Let $A, B \in \mathtt{T}$ such that $FV(A) \subseteq \{1, \ldots n\}$ and $A \to_\beta B$.
Then $u_{[x_n,\ldots,x_1]}(A) \to_\beta u_{[x_n,\ldots,x_1]}(B)$.

THEOREM 3.28
Let $\Gamma = A_1, \ldots, A_n$ such that $\Gamma \vdash A : B$.
Then $u(\Gamma) \vdash u_{[v_1,\ldots,v_n]}(A) : u_{[v_1,\ldots,v_n]}(B)$.

3.2.3 t and u are inverses

We need to check that in some sense $t \circ u = Id$ and $u \circ t = Id$. We begin by studying $t \circ u$, which as expected is exactly the identity. We prove first the following lemma.

LEMMA 3.29
Let $A \in \mathtt{T}$ such that $FV(A) \subseteq \{1, \ldots, n\}$ and let $x_1, \ldots, x_n$ be distinct variables. Then $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(A)) \equiv A$.

PROPOSITION 3.30
Let $\Gamma = A_1, \ldots, A_n$ such that $\Gamma \vdash A : B$. Then the derivations $\Gamma \vdash A : B$ and $t(u(\Gamma)) \vdash t_{[v_n,\ldots,v_1]}(u_{[v_1,\ldots,v_n]}(A)) : t_{[v_n,\ldots,v_1]}(u_{[v_1,\ldots,v_n]}(B))$ are exactly the same.

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 3.31
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:
$\pi <> .A =_{def} A \qquad$ and $\qquad \pi(\Gamma, x : B).A =_{def} \pi\Gamma.\pi x : B.A$
We say that the derivations $\Gamma \vdash A : B$ and $\Gamma' \vdash A' : B'$ are *equivalent* when $\lambda\Gamma.A \equiv \lambda\Gamma'.A'$ and $\Pi\Gamma.B \equiv \Pi\Gamma'.B'$.

LEMMA 3.32
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 $u_{[x_n,\ldots,x_1]}(t_{[x_1,\ldots,x_n]}(A)) \equiv A$.

PROPOSITION 3.33
Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $A,\, B \in \mathcal{T}$. The derivations
$\Gamma \vdash A : B$ and $u(t(\Gamma)) \vdash u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(A)) : u_{[v_1,\ldots,v_n]}(t_{[x_n,\ldots,x_1]}(B))$ are equivalent in the sense of Definition 3.31.

With the above isomorphism, we can now establish Theorems 3.11 and 3.13 for PTSs with de Bruijn indices.

THEOREM 3.34 (The Church Rosser Theorem for PTSs with de Bruijn indices)
In T, 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 Lemma 3.27, $u_{[x_n,\ldots,x_1]}(A) \twoheadrightarrow_\beta u_{[x_n,\ldots,x_1]}(B)$ and $u_{[x_n,\ldots,x_1]}(A) \twoheadrightarrow_\beta u_{[x_n,\ldots,x_1]}(C)$. Hence, by Theorem 3.11, $\exists D$ 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 $FV(u_{[x_n,\ldots,x_1]}(-)) \subseteq \{x_1, \ldots, x_n\}$ for $- \in \{B, C\}$ and hence by Lemma 3.23 we get $t_{[x_1,\ldots,x_n]}(u_{[x_n,\ldots,x_1]}(-)) \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$ for $- \in \{B, C\}$. Then, Lemma 3.19 sorts out the free variable condition for Lemma 3.29, and the latter gives $- \twoheadrightarrow_\beta t_{[x_1,\ldots,x_n]}(D)$ for $- \in \{B, C\}$. ∎

THEOREM 3.35 (Subject Reduction SR, for PTSs with de Bruijn indices)

In T, if $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$.

PROOF. First, we use Theorem 3.28 and Lemma 3.27 to obtain the conditions of Theorem 3.13 in $\mathcal{T}$. Then, we use Theorem 3.24 and Proposition 3.30 to obtain SR in T. ∎

Acknowledgements

Without the thorough reading and editing of Mariangiola Dezani, this review would not have been what it is now. I am extremely grateful for her work on this article and for her excellent recommendations. I am also very grateful to Joe Wells for his extensive and invaluable contributions to Section 1 of this review. The influence of the work of Henk Barendregt, N.G. de Bruijn and Jan Willem Klop is evident in this review; their high standards and decent and warm personalities make this school of thought a very challenging and enjoyable place. I would like to warmly thank Rob Nederpelt and Alejandro Ríos for the valuable collaboration I've had with them. This collaboration is evident in this review. This work was partially supported by EPSRC grant numbers GR/L36963 and GR/L15685.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, **1**, 375–416, 1991.

[2] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call by need lambda calculus. *22nd ACM Symposium on Principles of Programming Languages*, 1995.

[3] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics* (revised edition). North Holland, 1984.

[4] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume II, S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, eds. 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] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (Erster Artikel). *Mathematische Annalen*, **46**, 481–512, 1895.

[7] G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre (Zweiter Artikel). *Mathematische Annalen*, **49**, 207–246, 1897.

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

[9] T. Coquand and G. Huet, The calculus of constructions, *Information and Computation*, **76**, 95–120, 1988.

[10] H. B. Curry. *A Theory of Formal Deducibility*. Notre Dame Mathematical Lectures, 6. Notre Dame University Press, 1950.

[11] H. B. Curry and R. Feys. *Combinatory Logic I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[12] Euclid. *The Thirteen Books of Euclid's Elements*. Volume 1 (books I and II). Translated with introduction and commentary by Sir Thomas L. Heath. Second edition unabridged 1925. Reprinted by Dover.

[13] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Nebert, Halle, 1879. Also in [19], pp. 1–82.

[14] G. Frege. *Grundgesetze der Arithmetik, begriffschriftlich abgeleitet*, volume I. Pohle, Jena, 1892. Reprinted 1962 (Olms, Hildesheim).

[15] G. Frege. Letter to Russell. English translation in [19], pages 127–128, 1902.

[16] G. Frege. *Grundgesetze der Arithmetik, begriffschriftlich abgeleitet*, volume II. Pohle, Jena, 1903. Reprinted 1962 (Olms, Hildesheim).

[17] P. de Groote. The conservation theorem revisited. In *International Conference on Typed Lambda Calculi and Applications, LNCS*, volume 664. Springer-Verlag, 1993.

[18] B. Guillaume. *Un calcul des substitutions avec etiquettes*. PhD thesis, Université de Savoie, Chambéry, 1999.

[19] J. van Heijenoort, ed. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

[20] A. Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer-Verlag, Berlin, 1934.

[21] R.J. Hindley and J.P. Seldin *Introduction to Combinators and λ-calculus*, Cambridge Univeristy Press, 1986.

[22] W. A. Howard. The formulaes-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds. pp. 479–490. Academic Press, 1980. An earlier version was privately circulated in 1969.

[23] F. Kamareddine and R.P. Nederpelt. On stepwise explicit substitution. *International Journal of Foundations of Computer Science*, **4**, 197–240, 1993.

[24] F. Kamareddine and R.P. Nederpelt. Generalising reduction in the λ-calculus. *Journal of Functional Programming*, **5**, 637–651, 1995.

[25] F. Kamareddine and R.P. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, **155**, 85–109, 1996.

[26] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitutions. In *Proceedings of Programming Languages Implementation and the Logic of Programs PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pp. 45–62. Springer-Verlag, 1995.

[27] F. Kamareddine and A. Ríos. Pure Type Systems with de Bruijn indices. `http://www.cee.hw.ac.uk/~fairouz/papers/research-reports/ptsdebruijn.ps`. Submitted for publication.

[28] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, **7**, 395–420, 1997.

[29] M. Karr. Delayability in proofs of strong normalizability in the typed λ-calculus. In *Mathematical Foundations of Computer Software, LNCS*, volume 185. Springer-Verlag, 1985.

[30] A.J. Kfoury and J.B. Wells. New notions of reductions and non-semantic proofs of β-strong normalisation in typed λ-calculi. *LICS*, 1995.

[31] J. W. Klop. Combinatory Reduction Systems. *Mathematical Center Tracts*, 27, 1980. CWI.

[32] A. N. Kolmogorov. Zur Deutung der Intuitionistischen Logik. *Mathematisches Zeitschrift*, **35**, 58–65, 1932.

[33] T. Laan. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Eindhoven University of Technology, 1997.

[34] E. Magnusson. *The implementation of ALF - a proof editor based on Martin Löf's Type Theory with explicit substitutions*. PhD thesis, Chalmers, 1995.

[35] M. Mauny. *Compilation des langages fonctionnels dans les combinateurs caté-goriques. Application au langage ML*. PhD thesis, Université Paris VII, 1985.

[36] C. Muñoz. Proof representation in type theory: State of the art. In *Proceedings of the XXII Latin-American Conference of Informatics CLEI Panel 96*, Santafé de Bogotá, Colombia, June 1996.

[37] C. Muñoz. Proof synthesis via explicit substitutions on open terms. In *Proc. International Workshop on Explicit Substitutions, Theory and Applications, WESTAPP 98*, Tsukuba (Japan), April 1998.

[38] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. North-Holland, Amsterdam, 1994.

[39] G. Peano. *Arithmetices principia, nova methodo exposita*. Bocca, Turin, 1889. English translation in [19], pp. 83–97.

[40] L. Regnier. Une équivalence sur les lambda termes. *Theoretical Computer Science*, **126**, 281–292, 1994.

[41] A. Ríos. *Contribution à l'étude des λ-calculs avec substitutions explicites*. PhD thesis, Université de Paris 7, 1993.

[42] B. Russell. Letter to Frege. English translation in [19], pp. 124–125, 1902.

[43] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, **30**, 222–262, 1908. Also in [19], pp. 150–182.

[44] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pp. 288–298, 1992.

[45] M. M. Sørensen. Strong normalisation from weak normalisation in typed λ-calculi. *Information and Computation*, **133**, 35–71, 1997.

[46] M. Takahashi. Parallel reduction in λ-calculus. *Information and Computation*, **118**, 120–127, 1995.

[47] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmegen, 1989.

[48] D. Vidal. *Nouvelles notions de réduction en lambda calcul*. PhD thesis, Université de Nancy 1, 1989.

[49] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume I, II, III. Cambridge University Press, 1910^1, 1927^2. All references are to the first volume, unless otherwise stated.

[50] H. Xi. On weak and strong normalisations. Technical Report 96-187, Carnegie Mellon University, 1996.

Received 4 October 2000