A strong normalization result for classical logic

Franco Barbanera *, Stefano Berardi

Dipartimento di Informatica, Università di Torino, Corso Svizzera, 185 10149 Torino, Italy

Received 19 September 1993; revised 17 January 1995; communicated by D. van Dalen

Abstract

In this paper we give a strong normalization proof for a set of reduction rules for classical logic. These reductions, more general than the ones usually considered in literature, are inspired to the reductions of Felleisen's lambda calculus with continuations.

1. Introduction

Recently, in the logic and theoretical computer science community, there has been an ever growing interest in the computational features of classical logic. The problem on which research is beginning to focus now is not the theoretical possibility of having constructive content present in classical proofs, established in old and well-known results, but the practical applicability of such results.

It was Kreisel in [12], who first pinpointed the presence of constructive content in classical proofs by proving the equality of the sets of $\Sigma_1^0$-sentences provable, respectively, in intuitionistic and classical logic. Friedman [7] showed how to get the computational content of a classical proof of a $\Sigma_1^0$-sentence by means of a translation from classical to intuitionistic logic. Such method, however, can hardly be satisfactory since, to really use the computational features of classical logic, one needs to know how to extract directly computational content from proofs, i.e. to reduce a proof to its essential content. For intuitionistic logic this problem amounts to cut elimination, in particular to strong normalizability for systems in natural deduction. Unfortunately there is no good cut-elimination procedure for classical logic, i.e. speaking in term of natural deductions, no good set of reduction rules is known.

In literature some sets of reduction rules can be found for classical logic, as the one defined in [15] by Prawitz. This set of reductions can also be used to extract

* Corresponding author. E-mail: barba@di.unito.it

0168-0072/95/$09.50 © 1995 Elsevier Science B.V. All rights reserved
SSDI 0168-0072(95)00004-6
computational content from classical proofs as shown in [1], but can hardly be considered adequate for classical logic. Another set of rules is the one defined by Parigot for his μ-calculus [14].

In the quest of a good set of reductions for classical logic recent results by Griffin and Murthy shed a light in what seems to be a good direction. In [11] and [13] they showed that a proofs-as-programs correspondence can be defined between classical proofs and control functional languages. In particular, a classical proof can be seen as a typed term in a lambda calculus containing Felleisen’s control operator % (λ%) [4, 6] and constructive content can be got from a proof by reducing it using the reductions of the calculus. Such reductions are quite general from a logical point of view (for instance they subsume Prawitz’s reductions). Their most interesting feature is that there exists a sort of symmetry between some of them.

Unfortunately, such rules cannot be considered, strictly speaking, good logical rules, since, at the time being, no strong normalization result exists. In fact the possibility of using them to extract constructive content from classical proofs of \( \Sigma^0_1 \) sentences has been established only by restricting oneself to particular reduction strategies (i.e. only weak normalizations have been proved).

Notwithstanding the above-mentioned difficulties this set of reduction rules can be a good starting point to develop good “classical” reductions. In the present paper we define a set of reductions, inspired to the set of reductions for \( \lambda\varepsilon \), for a \( \lambda \)-calculus (\( \lambda\varepsilon\varepsilon \)) whose terms represent proofs of propositional classical logic in natural deduction. With respects to those for \( \lambda\varepsilon \), some reductions of \( \lambda\varepsilon\varepsilon \) are new, other are restrictions. They however maintain the feature of \( \lambda\varepsilon \)’s reductions mentioned above: symmetry.

For \( \lambda\varepsilon\varepsilon \) we manage to get a strong normalization proof. This proof takes the main part of the paper and consists in a nontrivial modification of Tait–Girard computability method. The nature of the reductions of our system makes it impossible to use the usual notion of computability, since otherwise a circularity would arise. In order to overcome this problem we stratify the notion of computability over an ordinal parameter, i.e. we consider it as a general inductive definition. Ordinal induction over this parameter will be essential in the strong normalization proof.

We have then a powerful and strongly normalizable set of logical reductions for classical logic whose strong normalization property can be got at the price of loosening the connection with system \( \lambda\varepsilon \). As evidence of the power of our reduction rules, in [2] it has been proved that they can be extended to first-order classical logic and be used to extract constructive contents from classical proofs of \( \Sigma^1_0 \)-sentences. Besides, the strong normalization property for the extended set of rules is shown in [2] to be an easy consequence of our strong normalization result for \( \lambda\varepsilon\varepsilon \).

2. The system \( \lambda\varepsilon\varepsilon \)

In this section we describe a typed system \( \lambda\varepsilon\varepsilon \) and a set of reduction rules on its terms. Types of \( \lambda\varepsilon\varepsilon \) correspond to propositional logical sentences and its terms to
proofs in classical logic. We have chosen to provide such a system since terms are easier to "handle" than proofs.

In the following we shall then use interchangeably the words "term" and "proof", as well as "formula" and "type".

The typing for our terms will follow the one proposed in [11, 13] for Felleisen's calculus $\lambda_{\&}$.

Mainly because of the technical motivations mentioned in the introduction and that we shall make clearer in the following, we restrict the types of $\lambda_{\&}$ to a strict subset of all the possible logical formulas, even if we shall see this not to be a real restriction (the superscript "-" on the $\tau$ in the name $\lambda_{\&}$, expressing in turn the "typefulness" of the system, is to recall that we do not consider all the possible types).

The types of our system are a subset of the simple types à la Church, i.e. of the types built out of atomic types $a, b, c, \ldots$ and using the connectives $\to$ (implication) and $\bot$ (falsehood). The negation in our system is defined as usual, by

$$\neg A \overset{\text{Def}}{=} A \to \bot.$$ 

We restrict these types by forbidding types to have proper subtypes of the form $\neg \neg A$. We also forbid $\bot$ to occur on the left-hand side of $\to$ (like in $\bot \to A$).

Hence, for instance, $\neg \neg A$ and $\neg (B \to \neg A)$ ($A, B \neq \bot$) are types of our system, in case $A$ and $B$ are so, while $A \to \neg B$ and $A \to (\bot \to B)$ can never be.

One could wonder whether the language of our system is rich enough. Indeed, even if our system, as shown below in the definition of the rules, contains the "ex falso quodlibet" and "double negation elimination" rules, $\bot \to A$ and $\neg \neg A \to A$ are not well-formed formulas. This, however, does not limit the sort of classical proofs it is possible to express in the system, since in meaningful classical reasoning only the corresponding rules are used. The same argument applies to the possible objection to the fact that no strict subformula in our system can be a double negation; in fact, in the common practice, a double negation would be useless inside formulas. It makes sense only outside them and before applying the "double negation elimination" rule.

So our calculus formally defines a fragment of classical logic, but its restrictions are the ones that are implicitly used in the common proving practice. Moreover, it is easy to write a procedure that, given a proof in classical logic not respecting our restrictions, modifies it in such a way the restrictions are respected, but without changing its sense. The system we define is then sufficient for proof-theoretical purposes.

The formal definition of the types of $\lambda_{\&}$ runs as follows:

**Definition 2.1.** The sets of Positive types ($P$), Negative types ($N$), PosNeg types ($U$) and Double negated types ($D$) are defined by the following grammars, where $a$ ranges over the set of type constants:

- $P ::= a | P \to P | \neg P | \neg P \to P | \neg P \to \neg P$
- $N ::= \neg P$
- $U ::= P | N$
- $D ::= \neg \neg P$. 
The set of Types (T) of $\lambda_{\mathcal{T}}$ is now defined by
\[
T := \bot | P | N | D.
\]

Then positive types are those which are not negations; $\bot$ can occur in a type only in subtypes of the form $\neg A$ and a double negation can occur outside, but not inside types.

In the following, Positive types will be denoted by $P, P', P'', \ldots$. Types and PosNeg types by $T, A, B, C, \ldots$. So, if $A$ is a PosNeg type, $\neg A$ denotes a correct type, maybe double negated, while $\neg \neg A$, instead, may be out of the set of correct types, in case $A$ be itself negated.

For each type $T$, we suppose to have infinitely many variables labelled with $T$:
\[
\text{Var}_T := \{x_0^T, x_1^T, x_2^T, \ldots\}
\]

We shall drop the label $T$ when it will be clear from the context.

We define now a set of "pseudoterms" and a set of typing rules. The terms of system $\lambda_{\mathcal{T}}$ will be the pseudoterms having a correct type. The pseudoterms are built out of variables, using abstraction, application and the operators $\mathcal{C}$ (which will correspond to double negation elimination for well-formed terms) and $\mathcal{A}$ (which will correspond to the ex falso quodlibet rule). We shall assume each occurrence of the operator $\mathcal{A}$ to have a type label $T \neq \bot (\mathcal{T})$ which we shall drop when unnecessary.

**Definition 2.2.** The set of Pseudoterms of $\lambda_{\mathcal{T}}$ is defined by the following grammar:
\[
M ::= x^T \mid \lambda x. M \mid (MM) \mid \mathcal{C}M \mid \mathcal{A}T M
\]

**Definition 2.3** (Typing rules). Let $A, B$ be PosNeg types, $P$ a positive type, $T$ a type $\neq \bot$ and $M, N$ pseudoterms.

\[
\begin{align*}
(\text{var}) & \quad x^T : T \\
(\rightarrow I) & \quad M : A \quad \frac{\lambda x^A . M : A \rightarrow B}{\lambda x^A . M : A \rightarrow B} \\
(\rightarrow E) & \quad \frac{M : A \rightarrow B \quad N : A}{MN : B} \\
(\neg I) & \quad M : \bot \quad \frac{\lambda x^A . M : \neg A}{\lambda x^A . M : \neg A} \\
(\neg E) & \quad \frac{M : \neg A \quad N : A}{MN : \bot} \\
(\neg \neg E) & \quad \frac{\mathcal{C}M : P}{\mathcal{A}M : P} \\
(\mathcal{A}) & \quad \frac{M : \bot}{\mathcal{A}T M : T}
\end{align*}
\]

We call then term a pseudoterm having a correct type.

The cases of introduction and elimination of $\neg$, even if it is a derived symbol, have been treated separately because of our type restrictions.
It is not difficult to see that the type of a term is unique (because of the type labels on variables) and may be computed.

We shall denote by $\text{Term}_T$ the set of terms having type $T$. A term of the form $\forall M$ will be called a dne-term (double negation elimination). One of the form $\exists M$ an efg-term (ex falso quodlibet).

We introduce now the reductions for terms of $\lambda_{\text{dne}}$.

**Definition 2.4 (Reduction rules).**

\[
\begin{align*}
(\beta) & \quad (\lambda x.M)N \rightarrow_1 M[N/x] \\
(\mathcal{C}_L) & \quad (\forall M)N \rightarrow_1 \forall \lambda k.M(\lambda f.k(fN)) \quad (1) \\
(\mathcal{C}_L') & \quad (\forall M)N \rightarrow_1 \lambda p.M(\lambda f.(fN)p) \quad (2) \\
(\mathcal{C}_R) & \quad M(\forall N) \rightarrow_1 \forall \lambda k.N(\lambda a.k(Ma)) \quad (3) \\
(\mathcal{C}_R') & \quad M(\forall N) \rightarrow_1 \lambda p.N(\lambda a.(Ma)p) \quad (4) \\
(\mathcal{C}_R'') & \quad M(\forall N) \rightarrow_1 N(\lambda a.(Ma)) \quad (5) \\
(\mathcal{A}) & \quad E[\mathcal{A}M] \rightarrow_1 M \quad (6)
\end{align*}
\]

Provisos: 
(1) $M$ has to have type of the form $\neg(\rightarrow(P \rightarrow Q))$
(2) $M$ has to have type of the form $\neg(\rightarrow(\rightarrow Q))$
(3) $M$ has to have type of the form $\rightarrow(\rightarrow Q)$
(4) $M$ has to have type of the form $\rightarrow(\rightarrow \neg Q)$
(5) $M$ has to have type of the form $\neg Q$
(6) $E[-]$ is a context $\neq [-]$ with type $\bot$ and $FV(M) \subseteq FV(E[\mathcal{A}M])$

$\rightarrow_1$ will denote the reflexive and transitive closure of $\rightarrow$.

The symbols of the above notions of reduction will be used in the following also to denote their compatible closure.

It can be noted that the reductions $\mathcal{C}_R$ and $\mathcal{C}_L$ are a typed version of the corresponding reductions for $\lambda_{\text{dne}}$ (the names $\mathcal{C}$ and $\mathcal{A}$ for our operators have been chosen exactly to recall such a fact), with the restriction that the redexes $(\forall M)N$ and $M(\forall N)$ must have a positive type. Such rules, by making an elimination of double negation to be applied on simpler and simpler formulas, enable the constructive content hidden in a proof to come out. When the type of $(\forall M)N$ or $M(\forall N)$ is negative, it is immediate to check that they cannot be applied because of the restrictions on the types of our system. In such case, however, one can notice that the use of the double negation elimination is not essential. In fact, if a proof contains double negation eliminations only in subterms of the form $(\forall M)N$ or $M(\forall N)$ with negative types, it can be transformed in a proof of intuitionistic logic. This observation has led to the definition of the additional rules $\mathcal{C}_R'$ and $\mathcal{C}_L'$, where the left-hand sides $(\forall M)N$ and $M(\forall N)$ have a negative type and the right-hand sides do not contain $\mathcal{C}$ and correspond to intuitionistic arguments.

Rule $\mathcal{C}_R''$ has been introduced in order to deal with the case of the elimination of negation. Also in this rule the use of the double negation elimination can be avoided in its right-hand side.
In the reductions defined above we have not put the type decorations for sake of readability. We give below the reduction rules with all the type decorations.

\( (\beta) \quad ((\lambda^{A}.M^{B})^{A\rightarrow B}N^{A})^{B} \rightarrow_{1} (M[N/x])^{B} \)

\( (\infty_{L}^{1}) \quad ((\infty(M)^{\neg(A\rightarrow P)}A\rightarrow P)N^{A})^{P} \)

\[ \rightarrow_{1} (\infty(\lambda^{P}N^{P})(M(\lambda^{A}\rightarrow P)(k(fN))^{\bot})^{(A\rightarrow B)^{\bot}})^{\neg P} \]

\( (\infty_{L}^{1}) \quad ((\infty(M)^{\neg(A\rightarrow P)}A\rightarrow P)N^{A})^{P} \)

\[ \rightarrow_{1} (\lambda^{P}N^{P})(M(\lambda^{A}\rightarrow P)((fN)^{\neg P}p)^{\bot})^{(A\rightarrow P)^{\bot}}^{\neg P} \]

\( (\infty_{R}) \quad (M^{A}\rightarrow P)((\infty(N)^{A})^{P})^{P} \)

\[ \rightarrow_{1} (\infty(M^{A})(k^{A}(Ma)^{P})^{\bot})^{(A\rightarrow P)^{\bot}} \]

\( (\infty_{R}^{1}) \quad (M^{A}\rightarrow P)((\infty(N)^{A})^{P})^{P} \)

\[ \rightarrow_{1} (\lambda^{P}N^{P})(Ma^{A})^{(A\rightarrow P)^{\bot}}^{\bot} \]

\( (\infty_{R}^{1}) \quad (M^{A}\rightarrow P)((\infty(N)^{A})^{P})^{P} \)

\[ \rightarrow_{1} (\lambda^{P}N^{P})(Ma^{A})^{(A\rightarrow P)^{\bot}}^{\bot} \]

\( (\alpha) \quad (E[(\alpha M^{A})^{T}])^{\bot} \rightarrow_{1} M^{A} \)

**Definition 2.5.** Let \( n \) be an integer, \( M \) a term and \( T \) a type.

(i) \( n \) is a bound for \( M \) if the reduction tree of \( M \) has a finite height \( \leq n \).

(ii) \( M \) strongly normalizes if it has a bound.

(iii) \( \text{SN}_{T} = \{ M \in \text{Term}_{T} \mid M \text{ strongly normalizes} \} \)

**Theorem 2.1 (Strong normalization).** For any type \( T \):

\[ \text{SN}_{T} = \text{Term}^{T}, \]

i.e. any term \( M \) of \( \lambda_{\mathfrak{e}_{1}}^{\tau} \) is strongly normalizable.

The rest of the paper will be devoted to the proof of the Strong Normalization theorem.

3. **Strong normalization for \( \lambda_{\mathfrak{e}_{1}}^{\tau} \)**

Our proof method of strong normalization is essentially a nontrivial modification of Tait–Girard computability method. We sketch briefly now why, even with the restriction on types, this method is not applicable directly as it is, and what are the modifications we made to it.

The computability method, first introduced by Tait [17] for simply typed \( \lambda \)-calculus and later improved by Girard to his method of "candidates of reducibility" [9, 10] for second-order \( \lambda \)-calculus, is based on a notion of "computability". Computable terms strongly normalizes. Thus proving strong normalization reduces to showing that each term is computable, a thing that is not difficult to prove by induction on the term. However, to define a notion of computability for \( \lambda_{\mathfrak{e}_{1}}^{\tau} \), in particular for \( dne \)-terms, is
not easy. The first attempt which would naturally come in mind for the definition of a
notion of computability for terms of \( \lambda_k \), would be the following:

1. a variable is computable outright.
2. \( \lambda x.M \) is computable if, for all computable terms \( N \) with the same type of \( x \),
   \( M[N/x] \) is computable.
3. \( \emptyset(M) \) is computable if \( M \) is computable.
4. \( MN \) is computable if it strongly normalizes, and all its reducts which are not
   applications are computable (i.e., they satisfy either (1) or (2) or (3)).

This definition is incorrect as stated. While (2) is a definition by induction on the
type of the term, (3) forces a circularity. By (3), the computable terms \( \emptyset(M) \) of type
\( A \) are defined from the computable terms of type \( \neg A \). The latters, by (2), are defined
from the computable terms of type \( \neg A \) and hence, by (2) again, from the computable
terms of type \( A \).

What we do in order to overcome such a problem is to break this cycle by stratifying
the above definition over an ordinal parameter, i.e. by considering it as a general
inductive definition and using this ordinal induction during the proof.

We build the set of computable terms for each type \( A \) in several steps, in order not
to put the terms of the form \( \emptyset M \) in the set all together. In the first step we put in
the set of computable terms of type \( A \) only the terms which are not \( dne \)'s. Then in
step \( \alpha + 1 \) we consider the terms produced in step \( \alpha \), and add to the computable terms
of type \( A \) all the terms \( \emptyset M \) such that \( M \) was introduced in the computable terms of
type \( \neg A \) at the \( \alpha \)th step. By Tarski theorem it is impossible to go on indefinitely in
adding terms; we have to stop at most at step \( \omega_1 \), the first uncountable ordinal. The
set obtained at such limit ordinal is then the set of computable terms of type \( A \).

The sets built as sketched above will be proved to be, following Girard's method,
candidates, i.e. sets of terms having certain properties, among which strong normal-
ization. As said before, it will not be difficult then to show that all terms are indeed
computable.

### 3.1. Stratified candidates for \( \lambda_k \)

In this section we define a notion of candidate for our language, associate to each
type a set of terms (the computable ones) and prove these sets to be candidates.

**Definition 3.1.** A set \( X \) of terms is a candidate for a type \( T \) iff the following conditions
hold:

- (C0) \( X \supseteq \text{Var}_T \)
- (C1) \( \text{SN}_T \supseteq X \)
- (C2) \( X \) is closed by reductions
- (C3) for every \( MN \in \text{Term}_T \): if \( (\forall Q \in \text{Term}_T)(MN \rightarrow Q \Rightarrow Q \in X) \) then \( MN \in X \).

**Lemma 3.1.** \( \text{SN}_\bot \) is a candidate for \( \bot \).
Proof. Straightforward. □

In order to associate a particular candidate to each type, we first define some operators on sets of terms: Lambda, Lambda\(^\neg\), Not, Ap, Clos, Cont, EFQ, having the following functionality:

- **Lambda:** \(\mathcal{P}(\text{Term}_A) \times \mathcal{P}(\text{Term}_B) \rightarrow \mathcal{P}(\text{Term}_{A \rightarrow B})\)
- **Lambda\(^\neg\), Not:** \(\mathcal{P}(\text{Term}_A) \rightarrow \mathcal{P}(\text{Term}_{A \rightarrow B})\)
- **Ap, Clos:** \(\mathcal{P}(\text{Term}_A) \rightarrow \mathcal{P}(\text{Term}_A)\)
- **Cont:** \(\mathcal{P}(\text{Term}_{A \rightarrow B}) \rightarrow \mathcal{P}(\text{Term}_P)\)
- **EFQ:** \(\mathcal{P}(\text{Term}_A)\)

**Definition 3.2.** Let \(A, B\) be types, \(X \in \mathcal{P}(\text{Term}_A)\), \(Y \in \mathcal{P}(\text{Term}_B)\), \(Z \in \mathcal{P}(\text{Term}_{A \rightarrow B})\). We define:

- **EFQ** \(\equiv \{MN \in \text{snl} \mid (\forall \xi \in X)(Q \in X) = s Q E X\}\)
- **Lambda** \(\equiv \{\xi \in \text{snl} \mid (\forall \xi \in X)(M [Q/\xi] E Y)\}\)
- **Lambda\(^\neg\)** \(\equiv \{\xi \in \text{snl} \mid (\forall \xi \in X)(M [Q/\xi] E \text{snl})\}\)
- **Ap** \(\equiv \{MN \in \text{snl} \mid (\forall \xi \in X)(Q \not application & MN \rightarrow Q \Rightarrow Q E X)\}\)
- **Clos** \(\equiv X \cup \text{Ap}(X)\)
- **Not** \(\equiv \text{Clos}(\text{Var}_{A \rightarrow B} \cup \text{EFQ}_A \cup \text{Lambda}_\neg(X))\)
- **Cont** \(\equiv \{\xi \in \text{snl} \mid M E Z\}\)

It is possible to see that Lambda and Lambda\(^\neg\) express the constructive meaning of lambda abstraction, since a lambda abstraction can indeed be seen as a function from terms to terms. Ap says that the meaning of a term \(MN\) depends on the constructive meaning of its reducts. The use of the operator Clos is to close a set of terms \(X\) under Ap. The operator Not translates the constructive meaning of the negation, while Cont expresses the fact that the constructive meaning of a term \(\xi M\) is nothing but the constructive meaning of \(M\).

It is not difficult to see that the operator Lambda\(^\neg\) is decreasing w.r.t. the set-theoretical inclusion order; Ap and Cont are, instead, increasing. From the observations above it easily follows that Not is decreasing and the composition of Not with itself (Not o Not) is increasing.

We are now ready to define, for each type \(T\), a candidate \([T]\) associated to it. \([T]\) will be the \(\omega_1\) limit of an increasing chain \([T]\), of subsets of \(\text{term}_T\), where \(\omega\) denote an ordinal and \(\omega_1\) is the first uncountable ordinal. For any \(\alpha\) we will first define \([T]_\alpha\) for positive types and then extend the definition to nonpositive ones by using the operator Not.

**Definition 3.3.** Let \(T\) be a type \(\neq \perp\).

(i) \([\perp] = \text{Def} \ \text{snl}\).

(ii) We define \([T]_\alpha \in \mathcal{P}(\text{Term}_T)\), for each ordinal \(\alpha\), as follows:
(a) $\alpha = 0$

1. If $T$ is an atomic type $a$:

$$[T]_0 = \text{Def } \text{Clos}(\text{Var}_T \cup \text{EFQ}_T)$$

2. If $T$ is $A \rightarrow B$, and $B \neq \bot$, assume to have already defined $[T']$ for all subtypes $T'$ of $T$:

$$[T]_0 = \text{Def } \text{Clos}(\text{Var}_T \cup \text{EFQ}_T \cup \text{Lambda}([A], [B]))$$

3. If $T = \neg A$:

$$[T]_0 = \text{Def } \text{Not}([A]_0)$$

(b) $\alpha = \gamma + 1$

1. If $T$ is positive:

$$[T]_{\gamma+1} = \text{Def } \text{Clos}([T]_\gamma \cup \text{Cont}([\neg T]_\gamma))$$

2. If $T = \neg A$:

$$[T]_{\gamma+1} = \text{Def } \text{Not}([A]_{\gamma+1})$$

(c) $\alpha$ is a limit ordinal $\beta$:

1. If $T$ is positive:

$$[T]_\beta = \text{Def } \bigcup_{\gamma < \beta} [T]_\gamma$$

2. If $T = \neg A$:

$$[T]_\beta = \text{Def } \text{Not}([A]_\beta)$$

(iii) We define $[T]$ as follows:

$$[T] = \text{Def } [T]_{\omega_1}.$$  

If $P$ is a positive type it is easy to check that the chain $[P]_\omega$ is increasing because $[P]_{\omega+1} = \text{Clos}([P]_\omega \cup \ldots \supseteq [P]_\omega$. Therefore, by Tarski's Fixed Point Theorem we get $[P] = [P]_{\omega_1} = [P]_{\omega_1+1}$. Thus, by putting $\alpha = \omega_1$ in the definition of $[P]_{\omega+1}$ we have that

$$[P] = \text{Clos}([P] \cup \text{Cont}([\neg P]))$$

If $T = \neg A$, by the same argument we have instead $[T] = \text{Not}([A])$.

**Definition 3.4.** Let $M$ be a term and $A$ its type. Following Tait, we shall call $M$ computable iff $M \in [A]$.

Later we shall prove that every term is computable and that, for each type $A$, $[A]$ is indeed a candidate. It will follow, by Cl, that every term strongly normalizes.

### 3.1.1. Proving candidate properties

In this subsection we shall check that $[T]$, previously defined, is a candidate for any type $T$.

In the following Lemmas 3.2–3.5 we shall prove relevant properties of the operators we introduced. Then we shall be able to prove (in Lemma 3.6) that for each type $T$ and ordinal $\alpha$, the set $[T]_\alpha$ (in particular $[T]$) is a candidate.
Lemma 3.2. Let $T$ be a type and $X \in \mathcal{P}(\text{Term}_T)$. Then: $X$ satisfies C0–C2 $\Rightarrow$ Clos$(X)$ is a candidate.

Proof. We check separately C0, …, C3 for Clos$(X)$. Recall that Clos$(X) = X \cup \text{Ap}(X)$.

(0) Clos$(X) \supseteq X \supseteq \text{Var}_T$.

(1) $SN_T \supseteq \text{Clos}(X)$ by definition of Ap and $SN_T \supseteq X$.

(2) Assume $M \in \text{Clos}(X)$ and $M \rightarrow N$ in order to prove $N \in \text{Clos}(X)$. Then either $M \in X$, or $M \in \text{Ap}(X)$. In the first case, we apply C2 to $X$ and $M \rightarrow N$ in order to deduce $N \in X$. Thus, $N \in \text{Clos}(X)$ because Clos$(X) \supseteq X$. In the second case, by definition of Ap we know that $M = M_1M_2 \in SN_T$, and that

$$
(\forall Q \in \text{Term}_A)(Q \text{ not application and } M_1M_2 \rightarrow Q) \Rightarrow Q \in X
$$

(1)

Suppose now $N$ be not an application. Then from $M_1M_2 \rightarrow N$ we deduce $N \in X$ by 1. We are so reduced to the first case. If $N$ is instead an application, then, since $(N \rightarrow Q) \Rightarrow (M_1M_2 \rightarrow Q)$, from 1 we conclude:

$$
(\forall Q \in \text{Term}_A)(Q \text{ not application and } N \rightarrow Q) \Rightarrow Q \in X
$$

Therefore, $N \in \text{Ap}(X)$ by definition of Ap. Thus, $N \in \text{Clos}(X)$ because Clos$(X) \supseteq \text{Ap}(X)$.

(3) Assume:

$$
(\forall Q \in \text{Term}_A)(MN \rightarrow Q) \Rightarrow Q \in \text{Clos}(X)
$$

(2)

in order to prove $MN \in \text{Clos}(X)$. It is indeed enough to prove $MN \in \text{Ap}(X)$. There are finitely many $Q$'s such that $MN \rightarrow Q$, say, $Q_1, \ldots, Q_n$. Since each $Q_i \in \text{Clos}(X)$, and Clos$(X)$ satisfies C1, then each $Q_i$ has a bound $n_i$. Therefore, $\max\{n_i + 1\}$ is a bound for $MN$, and $MN \in SN_A$. To prove $MN \in \text{Ap}(X)$, there is still left to check:

$$
(\forall Q \in \text{Term}_A)(Q \text{ not application and } MN \rightarrow Q) \Rightarrow Q \in X
$$

(3)

To prove 3, assume $Q$ is not an application, and $MN \rightarrow Q$. Then $Q \neq MN$. It follows that for some $Q'$ we have $MN \rightarrow Q'$, say, $Q_1, \ldots, Q_n$. By the assumption 2, $Q' \in \text{Clos}(X)$. Then $Q \in \text{Clos}(X)$ follows by $Q' \rightarrow Q$, because Clos$(X)$ satisfies C2. Since $Q$ is not an application, then $Q \notin \text{Ap}(X)$, and therefore $Q \in X$. □

Lemma 3.3. Let $A, B$ be positive or negative types, $C$ any type $\neq \bot$, $X \in \mathcal{P}(\text{Term}_A)$, $Y \in \mathcal{P}(\text{Term}_B), Z, Z' \in (\text{Term}_C)$.

(i) $X$ satisfies C0, $Y$ satisfies C1, C2 $\Rightarrow$ Lambda$(X, Y)$ satisfies C1, C2.

(ii) $X$ satisfies C0 $\Rightarrow$ Lambda$^-(X)$ satisfies C1, C2.

(iii) $(Z, Z'$ satisfy C1, C2) and (Z or $Z'$ satisfy also C0) $\Rightarrow$ $Z \cup Z'$ satisfies C0, C1 and C2.

(iv) Var$^c$ satisfies C0, C1 and C2.

(v) Var$^c \cup EFQ^c$ satisfies C0, C1 and C2.
Proof. (i) Let $\lambda x. M \in \Lambda(X,Y)$. We check C1 and C2 separately.

(C1) We have to prove that $\lambda x. M \in SN_{\bot}$. By C0, $x \in X$; by definition of $\Lambda(X,Y)$, it follows $M \in Y$; by C1, we deduce $M \in SN_{\bot}$. Since $\lambda x. M$ has not type $\bot$, it is not an $\alpha$-redex, and each reduction out of $\lambda x. M$ is indeed a reduction out of $M$. Besides, any bound for $M$ is a bound for $\lambda x. M$ as well. We conclude that $\lambda x. M \in SN_{\bot}$.

(C2) Assume $\lambda x. M \rightarrow N$ in order to prove $N \in \Lambda(X,Y)$. Since $\lambda x. M$ is not an $\alpha$-redex, each reduction out of $\lambda x. M$ is indeed a reduction on $M$; then $N = \lambda x. Q$ and $M \rightarrow Q$. Therefore, it is enough to check that, for every $S \in X$, $Q[S/x] \in Y$. By $\lambda x. M \in \Lambda(X,Y)$ it follows that $M[S/x] \in Y$. Since $M[S/x] \rightarrow Q[S/x]$, by applying C2 to $Y$ we conclude that $Q[S/x] \in Y$. Thus, $N \in \Lambda(X,Y)$.

(ii) Similar to point (i). We use the fact that $SN_{\bot}$ is a candidate.

(iii) If $Z \supset Var_c$, then $Z \cup Z' \supset Var_c$. If $SN_c \supseteq Z, Z'$, then $SN_c \supseteq Z \cup Z'$. Assume now $Z$ and $Z'$ be closed by reduction, $M \in Z \cup Z'$ and $M \rightarrow N$. If $M \in Z$ then $N \in Z$; if $M \in Z'$ then $N \in Z'$. In both cases, $N \in Z \cup Z'$.

(iv) Straightforward. We use the fact that no reduction is possible on a variable.

(v) By (iv) and (iii), $Var_c \cup EFQ_c$ satisfies C0, C1 and C2 if $EFQ_c$ satisfies C1 and C2. Thus, we have to prove C1, C2 for $EFQ_c$. Since $C \neq \bot$, $\alpha_c M$ is not an $\alpha$-redex, and each reduction sequence out of $\alpha_c M \in EFQ_c$ is a reduction sequence out of $M$. Then $EFQ_c$ satisfies C1 and C2 since $M \in SN_{\bot}$ for each $\alpha_c M \in EFQ_c$.  

Lemma 3.4. Let $A, B$ be positive or negative types, $a$ an atomic type, $X \in \mathcal{P}(\text{T}erm_A)$.

(i) $X$ satisfies CO $\Rightarrow$ Not($X$) is a candidate for $\neg A$.

(ii) $[a]_0$ is a candidate for $a$.

(iii) $[A], [B]$ are candidates for $A, B \Rightarrow [A \rightarrow B]_0$ is a candidate for $A \rightarrow B$.

Proof.

(i) Since $X$ satisfies C0, then $Lambda(\neg)(X)$ satisfies C1, C2 by Lemma 3.3(ii). By applying 3.3(v), (iii) in this order, we deduce that $Var_{\neg A} \cup EFQ_{\neg A} \cup Lambda(\neg)(X)$ satisfies C0–C2. Thus, by Lemma 3.2, Not($X$) = Clos($Var_{\neg A} \cup EFQ_{\neg A} \cup Lambda(\neg)(X)$) is a candidate.

(ii) By Lemma 3.3(v) and Lemma 3.2, $[a]_0 = Clos(Var_a \cup EFQ_a)$ is a candidate.

(iii) If $[A]$ and $[B]$ are candidates for $A$ and $B$, then $Lambda([A],[B])$ satisfies C1, C2 by Lemma 3.3(i). By applying 3.3(v), (iii) in this order, we deduce that $Var_{A-B} \cup EFQ_{A-B} \cup Lambda([A],[B])$ satisfies C0–C2. We conclude, by Lemma 3.2, that $[A \rightarrow B]_0 = Clos(Var_{A-B} \cup EFQ_{a-B} \cup Lambda([A],[B]))$ is a candidate. 

Lemma 3.5. Let $P$ be a positive type, $X \in \mathcal{P}(\text{T}erm_P)$.

(i) $X$ satisfies C0 $\Rightarrow$ Cont(Not(Not($X$))) satisfies C1, C2.

(ii) $[P]_x$ is a candidate $\Rightarrow$ $[P]_{x+1}$ is a candidate.

(iii) Let $\beta$ be a limit ordinal, $[P]_x$ a candidate for all $\alpha < \beta \Rightarrow [P]_\beta$ is a candidate.

Proof. (i) Assume that $X$ satisfies C0. Then, by applying Lemma 3.4(i) twice, Not(Not($X$)) is a candidate. In particular, if $M \in Not(Not(X))$ and $M \rightarrow N$, then
\[ M \in SN_{\rightarrow P} \text{ by C1 and } N \in \text{Not(Not}(X)) \text{ by C2. We check now that Cont(Not(Not}(X))) \text{ satisfies C1, C2.} \]

(C1) Each reduction on \( \mathcal{C}M \) is indeed a reduction on \( M \), because \( M \) cannot have type \( \neg \neg \perp \) and thus \( \mathcal{C}M \) is not an \( \mathcal{A} \)-redex. It follows that if \( \mathcal{C}M \in \text{Cont}(\text{Not(Not}(X))) \) then \( \mathcal{C}M \in SN_P \).

(C2) For the same motivation, if \( \mathcal{C}M \rightarrow N' \), then \( N' = \mathcal{C}M' \) and \( M \rightarrow M' \), and therefore \( M' \in \text{Not(Not}(X)) \), \( N' = \mathcal{C}M' \in \text{Cont}(\text{Not(Not}(X))) \).

(ii) Assume \( [P]_\alpha \) be a candidate.

Then, by point (i) above, \( \text{Cont}(\neg P) = \text{Cont}(\text{Not}(\text{Not}(P))) \) satisfies C1, C2. By Lemma 3.3(iii) we deduce that \( [P]_\alpha \cup \text{Cont}(\neg P) \) satisfies C0–C2. We conclude that \( [P]_{\alpha+1} = \text{Clos}([P]_\alpha \cup \text{Cont}(\neg P)) \) is a candidate by Lemma 3.2.

(iii) By definition, \( [P]_\beta = \bigcup_{\alpha < \beta} [P]_\alpha \). Therefore we have to prove that the union of a non-empty increasing chain of candidates satisfies C0, C1, C2, C3.

(C0) The condition \( [P]_\alpha \supseteq \text{Var}_P \) is clearly preserved under non-empty unions.

(C1) Similarly for \( SN_P \supseteq [P]_\alpha \).

(C2) Assume \( M \in \bigcup_{\alpha < \beta} [P]_\alpha \) and \( M \rightarrow N \). Then \( M \in [P]_\alpha \) for some \( \alpha \), and \( N \in [P]_\alpha \) by C2 for \( [P]_\alpha \). Thus, \( N \in \bigcup_{\alpha < \beta} [P]_\alpha \).

(C3) Assume

\[ (\forall Q \in \text{Term}_\alpha)(MN \rightarrow Q) \Rightarrow Q \in \bigcup_{\alpha < \beta} [P]_\alpha. \]

\( MN \) has a finite number of one-step reducts, say \( Q_1, \ldots, Q_n \). For each of them we have \( Q_i \in \bigcup_{\alpha < \beta} [P]_\alpha \). Therefore, \( Q_i \in [P]_{\alpha_i} \) for some \( \alpha_i < \beta \). Let \( \alpha' = \max\{\alpha_1, \ldots, \alpha_n\} < \beta \) (with \( \alpha' = 0 \) if \( n = 0 \)). Since \( [P]_\alpha \) is an increasing chain, then \( [P]_{\alpha'} \supseteq [P]_{\alpha_i} \) and thus \( Q_1, \ldots, Q_n \in [P]_{\alpha'} \). We apply C3 to \( [P]_{\alpha'} \) and we deduce \( MN \in [P]_{\alpha'} \). It follows \( MN \in \bigcup_{\alpha < \beta} [P]_\alpha \). \( \square \)

**Lemma 3.6.** Let \( T \) be a type and \( \alpha \) an ordinal. Then \( [T]_\alpha \) is a candidate. In particular, \( [T] \) is a candidate.

**Proof.** By induction on the definition of \( [T]_\alpha \), i.e., by principal induction on the number of arrows in \( T \), and by secondary induction on \( \alpha \). All the properties required in the inductive steps are in Lemma 3.1, Lemma 3.4(i)–(iii) and Lemma 3.5(ii), (iii). \( \square \)

We are ready to prove now, in the next section, that every term is computable.

### 3.2. Computability for terms of \( \lambda_\mathcal{A} \)

In order to prove that every term is computable, we have to check that all constructors of the language build computable terms from computable terms. For some connectives, this fact follows by the definition we have given. For variables it follows from the fact that \( [A] \) is a candidate and from C0.
Lemma 3.7. Let $A, B$ be positive or negative types, $P$ a positive type, $\alpha$ an ordinal, $\lambda x.M \in \text{Term}_{\neg A}$ and $\lambda x.N \in \text{Term}_{A \rightarrow B}$. Then:

(i) $(\forall Q \in [A]_\alpha)(M[Q/x] \in SN_{\perp}) \Rightarrow \lambda x.M \in [\neg A]_\alpha$,
(ii) $(\forall R \in [A])(N[R/x] \in [B]) \Rightarrow \lambda x.N \in [A \rightarrow B]$,
(iii) $M \in [\neg P] \Rightarrow \mathcal{C}M \in [P]$,
(iv) $M \in SN_{\perp} \Rightarrow \mathcal{A}_A M \in [A]_\alpha$.

Proof. (i) $[\neg A]_\alpha \supseteq \text{Lambda}^{-}(\{[A]_\alpha\})$, and $\lambda x.M \in \text{Lambda}^{-}(\{[A]_\alpha\})$ by definition of Lambda$^{-}$.

(ii) $[A \rightarrow B] \supseteq \text{Lambda}(\{[A],[B]\})$, and $\lambda x.N \in \text{Lambda}(\{[A],[B]\})$ by definition of Lambda.

(iii) $[P] = \text{Clos}([P] \cup \text{Cont}([\neg P])) \supseteq \text{Cont}([\neg P])$ by Tarski’s Theorem, and $\mathcal{C}M \in \text{Cont}([\neg P])$ by definition of Cont.

(iv) From the definition of $[A]_\alpha$ it is easy to check that $[A]_\alpha \supseteq \text{EFQ}_A$ and hence $\mathcal{A}_A M \in [A]_\alpha$ when $M \in SN_{\perp}$.

To check instead that $M \in [A \rightarrow B]$ and $N \in [A]$ imply $MN \in [B]$ is more difficult. It will justify the need for the heavy candidate machinery we introduced.

The difficulty in proving $MN \in [B]$ lies in the fact that $MN$ has a functional constructive meaning (it may be reduced by $\beta$) but also non-functional ones (it may be reduced by $\mathcal{C}_L$, $\mathcal{C}_R$, $\mathcal{C}_L'$, $\mathcal{C}_R'$, $\mathcal{C}_L''$). Suppose, for instance, that $M = \mathcal{C}M'$, and reduce $MN$ by $\mathcal{C}_L$ to $\mathcal{C}_L k \cdot M' (\lambda f \cdot k (f N))$. If we try to prove $\mathcal{C}_L k \cdot M' (\lambda f \cdot k (f N)) \in [B]$, after a while, because of the presence of $\lambda f \ldots (f N) \ldots$ in $\mathcal{C}_L k \cdot M' (\lambda f \cdot k (f N))$, we are reduced to prove $M'' N \in [B]$ for any $M'' \in [A \rightarrow B]$.

The situation seems to be hopeless; in an attempt to prove $MN \in [B]$, we are reduced to prove $M'' N \in [B]$ for all $M'' \in [A \rightarrow B]$. In other words, the reduction rules we have seem to give a cyclic definition of the constructive meaning of $MN$. The idea is to break this cycle, by saying that we introduced $M = \mathcal{C}M'$ in $[A \rightarrow B]$ after we introduced $M' \in [\neg (A \rightarrow B)]$, and $M'' \in [A \rightarrow B]$. This informal idea has been formalized in the definition $[P]_{\alpha+1} \triangleq \text{Def Clos}([P]_\alpha \cup \text{Cont}([P]_\alpha))$. This definition says that if we introduced $M'$ in $[\neg P]$ at the stage $\alpha$, then we introduced $\mathcal{C}M'$ in $[P]$ at the stage $\alpha + 1$.

The next lemma (Lemma 3.8) characterizes the terms of the form $\mathcal{C}M$ occurring in $[P]_\alpha$. Then we will check (Lemmas 3.9 and 3.10) that $M \in [\neg A]$, $N \in [A]$ imply $MN \in SN_{\perp}$, and finally (Lemma 3.12) that $M \in [A \rightarrow B]$, $N \in [A]$ imply $MN \in [B]$. The last two properties are not easy to prove, but are required in order to show that every term is computable.

Lemma 3.8. Let $P$ be a positive type and $\alpha$ an ordinal. Then:

$\mathcal{C}M \in [P]_\alpha \Leftrightarrow \exists \alpha' < \alpha.M \in [\neg P]_{\alpha'}$.

Proof. ($\rightarrow$) $M \in [\neg P]_{\alpha'}$ for some $\alpha' < \alpha \Rightarrow (\mathcal{C}M) \in [P]_{\alpha' + 1}$ and $\alpha' + 1 < \alpha \Rightarrow (\mathcal{C}M) \in [P]_\alpha$ (because the chain $[P]_\alpha$ is increasing).
We proceed by induction on $\alpha$. The case $\alpha = 0$ is trivially satisfied, since no term of the form $\varphi$ can belong to $[P]_0$.

In the case $\alpha = \alpha' + 1$, if $\varphi \in [P]_{\alpha' + 1}$ then, since $\varphi$ is not an application, either $\varphi \in [P]_{\alpha'}$ or $\varphi \in \text{Cont}([\neg P]_{\alpha'})$. In the first case we apply the induction hypothesis on $\alpha'$ obtaining $M \in [\neg P]_{\alpha''}$ for some $\alpha'' < \alpha' < \alpha' + 1$; in the second one, $M \in [\neg P]_{\alpha'}$ by definition of Cont, and $\alpha' < \alpha' + 1$, as we wished to show.

In the case $\alpha$ is a limit ordinal, if $\varphi \in \bigcup_{\alpha' < \alpha} [P]_{\alpha'}$ then $\varphi \in [P]_{\alpha'}$ for some $\alpha' < \alpha$. We apply the induction hypothesis on $\alpha'$ and obtain $M \in [\neg P]_{\alpha''}$ for some $\alpha'' < \alpha' < \alpha$. $\square$

We check now (in Lemmas 3.9 and 3.10) that $M \in [\neg A], N \in [A]$ imply $MN \in SN_{\perp}$.

**Lemma 3.9.** Let $P$ be a positive type and $\alpha$ an ordinal. Then:

(i) $M \in [\neg P], N \in [P] \Rightarrow MN \in SN_{\perp}$,

(ii) $M \in [\neg P], N \in [P] \Rightarrow MN \in SN_{\perp}$.

**Proof.** (i) By Lemma 3.6, $[\neg P]$ and $[P]$ are candidates. By C1, $M$ and $N$ have bounds $m$ and $n$. We prove now that $MN \in SN_{\perp}$ by induction on $m + n$. Since $SN_{\perp}$ is a candidate by Lemma 3.1, by C3 it is enough to prove: $(\forall Q \in \text{Term}_{\perp})(MN \rightarrow Q) \Rightarrow Q \in SN_{\perp}$.

Assume $MN \rightarrow Q$ in order to prove $Q \in SN_{\perp}$. Then there are four cases: either $Q = M_1N$ and $M \rightarrow_1 M_1$, or $Q = MN_1$ and $N \rightarrow_1 N_1$, or $Q = M'[N/x]$ and $M = \lambda x.M'$, or $Q = R$ and $MN = E[\varphi R]$ (by definition, $\varphi''$ cannot be applied). In the first case, $M_1$ has a bound $m_1 < m$ and $M_1 \in [\neg P]_x$ by C2. In the second one $N_1$ has a bound $n_1 < n$ and $N_1 \in [P]_x$ by C2. In both cases we apply the induction hypothesis and deduce $M_1N$ (or $M_1N_1$) $\in SN_{\perp}$, as required to prove. In the third case, $M = \lambda x.M' \in [\neg P]_x \in \text{Not}([\neg P]_x)$. Since $M$ is neither an application nor a variable nor an $efq$-term, then $M \in \text{Lambda}^{-}([\neg P]_x)$, and we conclude $M'[N/x] \in SN_{\perp}$ by definition of $\text{Lambda}^{-}$. In the fourth case $R$ is necessarily a subterm of $M$ or $N$. In both cases $R \in SN_{\perp}$ since $M$ and $N$ have bounds.

(ii) Straightforward by (i), putting $\alpha = \omega_1$. $\square$

**Lemma 3.10.** Let $P$ be a positive type and $\alpha$ an ordinal. Then:

(i) $M \in [\neg P], N \in [P] \Rightarrow MN \in SN_{\perp}$,

(ii) $M \in [\neg P], N \in [P] \Rightarrow MN \in SN_{\perp}$.

**Proof.** (i) By Lemma 3.6, $[\neg P]$ and $[P]$ are candidates. By C1, $M$ and $N$ have bounds $m$ and $n$. We prove now $MN \in SN_{\perp}$ by principal induction on $\alpha$ and secondary induction on $m + n$. Since $SN_{\perp}$ is a candidate by Lemma 3.1, by C3 it is enough to prove: $(\forall Q \in \text{Term}_{\perp})(MN \rightarrow Q) \Rightarrow Q \in SN_{\perp}$. Assume $MN \rightarrow Q$ in order to prove $Q \in SN_{\perp}$. There are five possible cases:

1. $Q = M_1N$ and $M \rightarrow_1 M_1$. Then $M_1$ has a bound $m_1 < m$ and $M_1 \in [\neg P]$ by C2. By the secondary induction hypothesis we deduce $M_1N \in SN_{\perp}$.
2. $Q = MN_1$ and $N \rightarrow_1 N_1$. Then $N_1$ has a bound $n_1 < n$ and $N_1 \in [P]_\alpha$ by C2. By the secondary induction hypothesis we deduce $MN_1 \in SN_\perp$.

3. $Q = M'[N/x]$ and $M = \lambda x.M'$ (we applied $\beta$). Then $M \in \text{Lambda}^\perp([P])$, because $M \in [P]$ and $M$ is neither an application nor a variable nor an $efq$-term. We conclude $M'[N/x] \in SN_\perp$ by definition of $\text{Lambda}^\perp$, $N \in [P]_\alpha$ and $[P] \supseteq [P]_\alpha$ (the inclusion holds because $P$ is positive).

4. $Q = N'(\alpha a.(Ma))$ and $N = \varepsilon N'$ (we applied $\varepsilon''_R$). Then $N' \in [\neg P]_{\alpha'}$ for some $\alpha' < \alpha$, by Lemma 3.8 and $N \in [P]_\alpha$. To prove $N'(\lambda a.(Ma)) \in SN_\perp$, by Lemma 3.9 it is enough to prove $\lambda a.(Ma) \in [\neg P]_{\alpha'}$. By Lemma 3.7(i), $\lambda a.(Ma) \in [\neg P]_{\alpha'}$ may be proved if we prove $(\forall N'' \in [P]_{\alpha'})(MN'' \in SN_\perp)$. This last statement follows by principal inductive hypothesis on $\alpha' < \alpha$.

5. $Q = R$ and $MN = E[A \mathcal{R}]$ (we applied $\mathcal{A}$). Then $R$ is necessarily a subterm of $M$ or $N$. In both cases $R \in SN_\perp$ since $M$ and $N$ have bounds.

(ii) Straightforwardly by (i), putting $\alpha = \omega_1$. □

**Lemma 3.11.** For any positive or negative type $A$,

$$M \in [-A], N \in [A] \Rightarrow MN \in SN_\perp.$$  

**Proof.** By Lemma 3.9 (if $A$ is negative) or 3.10 (if $A$ is positive). □

We prove now the last and most difficult lemma of this paper.

**Lemma 3.12.** Let $A, B$ be positive or negative types and $\alpha, \beta$ be ordinals. If $A$ is a negative type, assume also $\beta = \omega_1$. Then,

(i) $M \in [A \rightarrow B]_\alpha, N \in [A]_\beta \Rightarrow MN \in [B],$

(ii) $M \in [A \rightarrow B], N \in [A] \Rightarrow MN \in [B].$

**Proof.** (i) By Lemma 3.6, $[A \rightarrow B]_\alpha$, $[A]_\beta$ and $[B]$ are candidates. By C1, $M$ and $N$ have bounds $m$ and $n$. We prove now $MN \in [B]$ by threefold induction on the indexes $\alpha, \beta, m + n$. (Actually, the order between the first two indexes does not matter.) By C3 it is enough to prove:

$$(\forall Q \in \text{Term}_B)(MN \rightarrow_1 Q) \Rightarrow Q \in [B].$$

Assume $MN \rightarrow_1 Q$ in order to prove $Q \in [B]$. There are seven possible cases:

1. $Q = M_1 N$ and $M \rightarrow_1 M_1$. Then $M_1$ has a bound $m_1 < m$ and $M_1 \in [A \rightarrow B]_\alpha$ by C2. By the induction hypothesis on $(m_1 + n)$ we deduce $M_1N \in [B]$.

2. $Q = MN_1$ and $N \rightarrow_1 N_1$. Then $N_1$ has a bound $n_1 < n$ and $N_1 \in [A]_\beta$ by C2. By the induction hypothesis on $(m + n_1)$ we deduce $MN_1 \in [B]$.

3. $Q = M'[N/x]$ and $M = \lambda x.M'$ (we applied rule $\beta$). Then $M \in \text{Lambda}([A], [B])$, because $M \in [A \rightarrow B]_\alpha$ and $M$ is neither an application nor a variable nor an $efq$- nor a $dne$-term. We conclude $M'[N/x] \in [B]$ by definition of $\text{Lambda}$. and $N \in [A]_\beta$, $[A] \supseteq [A]_\beta$. Remark that $[A] \supseteq [A]_\beta$ holds because either $A$ is a positive type (and the chain $[A]_\alpha$ is increasing) or $\alpha = \omega_1$ (and $[A] = [A]_\beta$).
4. Assume $Q = \lambda k.M'(\lambda f.k(fN))$, $M = \mathcal{G}M'$ and $B$ positive (we applied rule $\mathcal{G}_k$). Then $M' \in \neg\neg(A \rightarrow B)$ for some $\alpha' < \alpha$, by Lemma 3.8 and $M \in [A \rightarrow B]_\alpha$. We proceed now backwards from our thesis. To prove $\lambda k.M'(\lambda f.k(fN)) \in \{B\}$, by Lemma 3.7(iii) it is enough to prove $\lambda k.M'(\lambda f.k(fN)) \in \{\neg\neg\}$. By Lemma 3.11 and $M' \in \neg\neg(A \rightarrow B)_{\alpha'}$, $M'(\lambda f.R(fN)) \in SN_\perp$ may in turn be proved if we prove $\lambda f.R(fN) \in \{\neg\neg(A \rightarrow B)\}_{\alpha'}$. This last statement, by Lemma 3.7(i), is implied by $(\forall M'' \in [A \rightarrow B]_{\alpha'})(R(M''N) \in SN_\perp)$. Since $R \in \{\neg\}$, by Lemma 3.11 all we have to prove is $(\forall M'' \in [A \rightarrow B]_{\alpha'})(M''N \in \{B\})$. This last statement may be obtained by the principal induction hypothesis on $\alpha' < \alpha$.

5. $Q = \lambda p.M'(\lambda f.(fN)p)$ and $M = \mathcal{G}M'$, with $B = \neg P$ for some positive type $P$ (we applied rule $\mathcal{G}_k$). Then $M' \in \neg\neg(A \rightarrow B)_{\alpha'}$ for some $\alpha' < \alpha$, by Lemma 3.8 and $M \in [A \rightarrow B]_\alpha$. We proceed now backwards from our thesis. To prove $\lambda p.M'(\lambda f.(fN)p) \in \{B\} = \{\neg\neg P\}$ by Lemma 3.7(i) it is enough to prove $\lambda p.M'(\lambda f.(fN)p) \in \{\neg\neg\}$. By Lemma 3.11 and $M' \in \neg\neg(A \rightarrow B)_{\alpha'}$, $M'(\lambda f.R(fN)) \in SN_\perp$ may in turn be proved if we prove $\lambda f.R(fN) \in \{\neg\neg(A \rightarrow B)\}_{\alpha'}$. This last statement, by Lemma 3.7(i), is implied by $(\forall M'' \in [A \rightarrow B]_{\alpha'})(R(M''N) \in SN_\perp)$. Since $R \in \{\neg\}$, by Lemma 3.11 all we have to prove is $(\forall M'' \in [A \rightarrow B]_{\alpha'})(M''N \in \{B\})$. This last statement may be obtained by the principal induction hypothesis on $\alpha' < \alpha$.

6. $Q = \lambda k.N'(\lambda a.k(Ma))$, $N = \mathcal{G}N'$ and $B$ positive (we applied rule $\mathcal{G}_k$). Then $N' \in \neg\neg A_{\beta'}$ for some $\beta' < \beta$, by Lemma 3.8 and $N \in [A]_{\beta}$. Remark that, for the restriction we put on typing, $A$ must be a positive type, and therefore we did not assume $\beta = \omega_1$. We proceed now backwards from our thesis. To prove $\lambda k.N'(\lambda a.k(Ma)) \in \{B\}$ by Lemma 3.7(iii) it is enough to prove $\lambda k.N'(\lambda a.k(Ma)) \in \{\neg\neg\}$. By Lemma 3.7(i), $\lambda k.N'(\lambda a.k(Ma)) \in \{\neg\neg\}$ may be proved if we prove $(\forall R \in [P])(N'(\lambda a.R(Ma)) \in SN_\perp)$. By Lemma 3.9 and $N' \in \neg\neg A_{\beta'}$, $N'(\lambda a.R(Ma)) \in SN_\perp$ may in turn be proved if we prove $\lambda a.R(Ma) \in \{\neg\neg A\}_{\beta'}$. This last statement, by Lemma 3.7(i), may be deduced from $(\forall N'' \in [A]_{\beta'})(R(M''N) \in SN_\perp)$. Since $R \in \{\neg\}$, by Lemma 3.11 all we have to prove is $(\forall N'' \in [A]_{\beta'})(MN'' \in \{B\})$. This last statement may be obtained by the secondary induction hypothesis on $\beta' < \beta$ (in order to apply inductive hypothesis to $\beta' < \beta$, it is crucial that we did not assume $\beta = \omega_1$).

7. $Q = \lambda p.N'(\lambda a.(Ma)p)$ and $N = \mathcal{G}N'$ with $B = \neg P$ for some positive type $P$ (we applied rule $\mathcal{G}_k$). Then $N' \in \neg\neg A_{\beta'}$ for some $\beta' < \beta$, by Lemma 3.8 and $N \in [A]_{\beta}$. Remark that, for the restriction we put on typing, $A$ must be a positive type, and therefore we did not assume $\beta = \omega_1$. We proceed now backwards from our thesis. To prove $\lambda p.N'(\lambda a.(Ma)p) \in \{B\} = \{\neg\neg P\}$ by Lemma 3.7(i) it is enough to prove $(\forall R \in [P])(N'(\lambda a.(Ma)p) \in SN_\perp)$. By Lemma 3.9 and $N' \in \neg\neg A_{\beta'}$, $N'(\lambda a.(Ma)p) \in SN_\perp$ may in turn be proved if we prove $\lambda a.(Ma)p \in \{\neg\neg A\}_{\beta'}$. This last statement, by Lemma 3.7(i), may be deduced from $(\forall N'' \in [A]_{\beta'})(MN'' \in SN_\perp)$. Since $R \in \{\neg\}$, by Lemma 3.11 all we have to prove is $(\forall N'' \in [A]_{\beta'})(MN'' \in \{B\})$. This last statement may be obtained by the secondary induction hypothesis on $\beta' < \beta$. 
(in order to apply inductive hypothesis to $\beta' < \beta$, it is crucial that we did not assume $\beta = \omega_1$).

(ii) Straightforwardly by (i), putting $\alpha = \beta = \omega_1$. □

3.3. The result

We are now ready to prove a Soundness Theorem and to deduce Strong Normalization from it. We only need a last definition before.

Definition 3.5. Let $M$ be any term.

(i) A substitution is any map from a finite set of variables to the set of terms.

(ii) A substitution $\sigma$ is on $M$ if the free variables of $M$ are all in the domain of $\sigma$.

(iii) If $\sigma$ is a substitution on $M$, we denote by $\sigma(M)$ the result of replacing each $x$ free in $M$ by $\sigma(x)$.

(iv) A substitution $\sigma$ is computable if $\sigma(x)$ is computable for all variables $x$ in the domain of $\sigma$.

Theorem 3.1 (Soundness). Let $M$ be any term and $\sigma$ a substitution on it. Then:

$\sigma$ is computable $\Rightarrow \sigma(M)$ is computable.

Proof. By induction on $M$.

• $M$ is a variable. The thesis holds by definition of computable substitution.

• $M \equiv \bar{x}.M_1$. We apply Lemma 3.7(i) or 3.7(ii), if the type of $M$ has the form $\neg A$ or $A \rightarrow B$, respectively.

• $M \equiv M_1 \cdot M_2$. We apply Lemma 3.11 or 3.12, if the type of $M_1$ has the form $\neg A$ or $A \rightarrow B$, respectively.

• $M \equiv \bar{e}M_1$. We apply Lemma 3.7(iii).

• $M \equiv \bar{e}M_1$. We apply Lemma 3.7(iv). □

The Strong Normalization theorem turns now to be a corollary of the Soundness theorem.

Corollary 3.1 (Strong normalization). Every term $M$ of $\bar{e}$ strongly normalizes.

Proof. Let us consider the identical substitution $id$ on $M$, defined by $id(x) = x$ for any $x$ free in $M$. $id$ is computable because $x^4 \in [A]$ by C0. Therefore, by the Soundness Theorem, $M(= id(M))$ is computable. Thus, $M \in [A]$ for the type $A$ of $M$. Then, by Lemma 3.6 and C1, $M$ strongly normalizes. □

Acknowledgements

We are grateful to Mariangiola Dezani for her constant support and gentle guidance, and to Mario Coppo for helpful discussions and his careful reading of an earlier draft.
The first author wishes to express his gratitude also to Paola Francia for her steadfast encouragement.

References