

A general method for proving the normalization theorem for first and second order typed λ -calculi

VENANZIO CAPRETTA and SILVIO VALENTINI

*Dipartimento di Matematica Pura ed Applicata, Università di Padova,
via G. Belzoni n.7, I-35131 Padova, Italy*
Email: cprvnn12@leonardo.math.unipd.it and silvio@math.unipd.it

Received 23 June 1997; revised 20 March 1999

In this paper we describe a method for proving the normalization property for a large variety of typed lambda calculi of first and second order, which is based on a proof of equivalence of two deduction systems. We first illustrate the method on the elementary example of simply typed lambda calculus, and then we show how to extend it to a more expressive dependent type system. Finally we use it to prove the normalization theorem for Girard's system F.

1. Introduction

We will show a uniform method for proving the normalization property for a wide class of typed lambda calculi. It applies to many of the calculi that can be found in the literature (see, for example, Barendregt (1992)), for instance simply typed lambda calculus (whose normalization proof goes back to Turing (Gandy 1980)) and Girard's system F (Girard 1971).

Given a typed lambda calculus Λ , the method consists of three steps. The first step is to define a partially correct normalization algorithm nf_Λ for terms of Λ , then the second step is to find out a new calculus Λ_2 on whose terms nf_Λ terminates, and finally the last step is to prove that Λ and Λ_2 are equivalent. Thus, even if the normalization theorem is not new for the typed lambda calculi that we will analyze, for each of them we will give a new presentation that allows us a better understanding of its constructive content. This result is of particular interest for system F, whose standard presentation is completely impredicative, and which was the basic motivation for our work.

The paper is organized as follows. We first illustrate the method on simply typed lambda calculus: most of this section is already contained in Valentini (1994) and it was mainly inspired by some ideas of Tait and Martin-Löf. Then we show how to extend it to more expressive lambda calculi, for example, those obtained by introducing dependent and product types. Finally, we use the method to prove the normalization property for Girard's system F.

2. Normalization of simply typed lambda calculus

We begin by illustrating the method on simply typed lambda calculus. In order to fix our notation, we recall its rules in Appendix A.

The notion of normal form is central in the development of a typed lambda calculus: it establishes a canonical form for its terms, i.e. the *simplest* form. In the case we are considering, only one form of simplification is introduced, that is the relation obtained by closing for the term formation operations the following contraction relation.

Definition 2.1 (β -contraction).

$$(\lambda x:\alpha.b)(a) \rightsquigarrow b[x := a]$$

A term of the form $(\lambda x:\alpha.b)(a)$ is called a *redex*. A term is of the simplest form if it contains no redex[†].

Definition 2.2 (Normal form). A term is in normal form if it contains no redex.

2.1. A normalization algorithm

The normalization algorithm given below in this section, provided it is totally correct, shows a method for obtaining, for any given term, an equal term in normal form. It works on typed terms and its execution steps depend on the type of the term. This is the reason why we need a preliminary definition.

Definition 2.3. Let Γ be a context, t_1, \dots, t_n be a sequence of typed terms, α be a type and k be a natural number such that $1 \leq k \leq n$. Then, we define the k -th *argument type* of α with respect to t_1, \dots, t_n (denoted by $\alpha^{t_1, \dots, t_n; k}$) by induction on the type complexity of α , as follows:

- If $\alpha \equiv C$, then $\alpha^{t_1, \dots, t_n; k}$ is undefined.
- If $\alpha \equiv \beta \rightarrow \gamma$ and t_1 is a term of type β in the context Γ , then

$$\begin{aligned} \alpha^{t_1, \dots, t_n; 1} &= \beta \\ \alpha^{t_1, \dots, t_n; k+1} &= \gamma^{t_2, \dots, t_n; k}, \end{aligned}$$

otherwise $\alpha^{t_1, \dots, t_n; k}$ is undefined.

We will not explicitly write the context Γ in the notation $\alpha^{t_1, \dots, t_n; k}$ to make it more readable – the context will always be clear when we need to use the definition of *argument type*.

The normalization algorithm will take three arguments: a context Γ , a term e and a type α . In the description of the algorithm, supposing Γ is a context and x a variable in such a context, we will denote its type by Γ_x . Note that if Γ is a correct context a variable can be assumed at most once, so Γ_x is well defined.

[†] Here we consider only β -reduction and β -normal form, even if η -equality appears among the equality rules. This should not be considered a drawback of our method since the $\beta\eta$ -normal form theorem is a straightforward consequence of the β -normal form theorem.

Algorithm (Conversion into normal form)

Let Γ be a context, α be a type and e be a term. Then consider the following recursive definition:

$$\text{nf}(e^{\Gamma;\alpha}) = \begin{cases} (\lambda x:\beta.\text{nf}(e(x)^{\Gamma;x:\beta;\gamma})) & \text{if } \alpha \equiv \beta \rightarrow \gamma \text{ and} \\ & x:\beta \text{ is a fresh variable} \\ x(\text{nf}(t_1^{\Gamma;\Phi_1}) \dots (\text{nf}(t_n^{\Gamma;\Phi_n}))) & \text{if } \alpha \equiv C, \\ & e \equiv x(t_1) \dots (t_n) \text{ and} \\ & \Phi \equiv t_1, \dots, t_n \\ \text{nf}(c[x := a](t_1) \dots (t_n)^{\Gamma;\alpha}) & \text{if } \alpha \equiv C \text{ and} \\ & e \equiv (\lambda x:\beta.c)(a)(t_1) \dots (t_n). \end{cases}$$

The proof of partial correctness of this algorithm is easy.

Theorem 2.4 (Partial correctness). Let $\Gamma \vdash_{\lambda} e : \alpha$ and suppose the execution of $\text{nf}(e^{\Gamma;\alpha})$ terminates. Then $\text{nf}(e^{\Gamma;\alpha})$ is a term of type α in normal form equal to e .

Proof. Provided the algorithm of conversion into normal form terminates, the result is almost obvious. In fact, by induction on the number of steps in the execution of the algorithm it can be proved both that $\text{nf}(e^{\Gamma;\alpha})$ is a term of type α containing no redex and that e and $\text{nf}(e^{\Gamma;\alpha})$ are equal terms. \square

2.2. A new system to construct terms

The problem rests with proving the termination of the previous algorithm of conversion into normal form. To this aim we define a new system to derive terms such that a term is introduced only after the construction of those terms that are needed in order to prove its normalizability by means of our algorithm. In order to distinguish the terms of the new system from those of the old one, we will write $\Gamma \vdash_{\lambda_2} a : \alpha$, instead of $\Gamma \vdash_{\lambda} a : \alpha$, to mean that a is a term of type α in the context Γ .

Variable introduction

$$\frac{\Gamma \vdash_{\lambda_2} a_1 : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda_2} a_n : \alpha_n}{\Gamma, x:\alpha_1 \rightarrow \dots \alpha_n \rightarrow C \vdash_{\lambda_2} x(a_1) \dots (a_n) : C} \quad n \geq 0$$

 λ -introduction

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2} c : \gamma \quad \Gamma \vdash_{\lambda_2} a : \alpha \quad \Gamma \vdash_{\lambda_2} c[x := a](a_1) \dots (a_n) : C}{\Gamma \vdash_{\lambda_2} (\lambda x:\alpha.c)(a)(a_1) \dots (a_n) : C}$$

Abstraction

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2} b(x) : \beta}{\Gamma \vdash_{\lambda_2} b : \alpha \rightarrow \beta}$$

where the only free occurrence of x in $b(x)$ is the manifested one.

It is an immediate consequence of the way the new system is defined that the algorithm of normalization of the previous section terminates if applied to any one of the new terms.

2.3. Equivalence between the two systems

Now, to conclude the proof of normalization for any term of the original system, one must only show that it can be typed in the new system, that is, that it can be derived by means of the new rules.

First, note that it is easy to use induction on the structural complexity of the type of the considered variable to prove the following lemma, which states the closure of the new system under the variable introduction rule.

Lemma 2.5 (Closure under the variable introduction rule). Let α be a type. Then $x:\alpha \vdash_{\lambda_2} x : \alpha$.

The next step is to prove closure of the new system under substitution.

Lemma 2.6 (Closure under substitution). If $\Gamma, x:\alpha \vdash_{\lambda_2} b : \beta$ and $\Gamma \vdash_{\lambda_2} a : \alpha$ are derivable judgments, then $\Gamma \vdash_{\lambda_2} b[x := a] : \beta$ is a derivable judgment.

Proof. The proof is obtained by principal induction on the structural complexity of the type α of the substituted variable and secondary induction on the length of the derivation of the judgment $\Gamma, x:\alpha \vdash_{\lambda_2} b : \beta$. The only case in which the principal induction hypothesis is needed is the case of the variable introduction rule. In all other cases the proof is straightforward and only the secondary induction hypothesis is needed. For this reason, we will illustrate only the former case here.

Let us first consider the simpler case in which the variable z introduced by the rule is different from the variable x , that is, suppose that the instance of the rule is

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2} a_1 : \alpha_1 \quad \dots \quad \Gamma, x:\alpha \vdash_{\lambda_2} a_n : \alpha_n}{\Gamma, x:\alpha, z:\alpha_1 \rightarrow \dots \alpha_n \rightarrow C \vdash_{\lambda_2} z(a_1) \dots (a_n) : C}$$

In this case, by the induction hypothesis on the depth of the derivation, we know that, for $i = 1, \dots, n$, we have $\Gamma \vdash_{\lambda_2} a_i[x := a] : \alpha_i$ is a derivable judgment. Hence $\Gamma, z:\alpha_1 \rightarrow \dots \alpha_n \rightarrow C \vdash_{\lambda_2} z(a_1[x := a]) \dots (a_n[x := a]) : C$, which is identical to $\Gamma, z:\alpha_1 \rightarrow \dots \alpha_n \rightarrow C \vdash_{\lambda_2} z(a_1) \dots (a_n)[x := a] : C$, is derivable by using the variable introduction rule.

Suppose now that the variable introduced by the rule is exactly the substituted variable x . Two cases are possible: α is a basic type, and in this case the result is straightforward, or $\alpha \equiv \alpha_1 \rightarrow \dots \alpha_n \rightarrow C$, that is, the instance of the rule is

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2} a_1 : \alpha_1 \quad \dots \quad \Gamma, x:\alpha \vdash_{\lambda_2} a_n : \alpha_n}{\Gamma, x:\alpha_1 \rightarrow \dots \alpha_n \rightarrow C \vdash_{\lambda_2} x(a_1) \dots (a_n) : C}$$

As above, by the induction hypothesis on the depth of the derivation, we know that, for $i = 1, \dots, n$, we have $\Gamma \vdash_{\lambda_2} a_i[x := a] : \alpha_i$ is a derivable judgment. Observe now that the last rule used in the derivation of the judgment $\Gamma \vdash_{\lambda_2} a : \alpha_1 \rightarrow \dots \alpha_n \rightarrow C$ must have been an instance of the abstraction rule

$$\frac{\Gamma, y_1:\alpha_1 \vdash_{\lambda_2} a(y_1) : \alpha_2 \rightarrow \dots \alpha_n \rightarrow C}{\Gamma \vdash_{\lambda_2} a : \alpha_1 \rightarrow \dots \alpha_n \rightarrow C}$$

for some fresh variable y_1 . Since the type α_1 of the variable y_1 is simpler than α , by the principal induction hypothesis on the structural complexity of α , the judgment

$$\Gamma \vdash_{\lambda_2} a(a_1[x := a]) \equiv a(y_1)[y_1 := a_1[x := a]] : \alpha_2 \rightarrow \dots \alpha_n \rightarrow C$$

is derivable. The last rule in its derivation must in turn have been an instance of the abstraction rule

$$\frac{\Gamma, y_2 : \alpha_2 \vdash_{\lambda_2} a(a_1[x := a])(y_2) : \alpha_3 \rightarrow \dots \alpha_n \rightarrow C}{\Gamma \vdash_{\lambda_2} a(a_1[x := a]) : \alpha_2 \rightarrow \dots \alpha_n \rightarrow C}$$

for some fresh variable y_2 . Hence, again by the principal induction hypothesis, the judgment

$$\Gamma \vdash_{\lambda_2} a(a_1[x := a])(a_2[x := a]) : \alpha_3 \rightarrow \dots \alpha_n \rightarrow C$$

is derivable.

After n similar steps, we obtain that $\Gamma \vdash_{\lambda_2} a(a_1[x := a]) \dots (a_n[x := a]) : C$ is a derivable judgment, and that it is identical to $\Gamma \vdash_{\lambda_2} x(a_1) \dots (a_n)[x := a] : C$. \square

Now the missing link can easily be established.

Theorem 2.7 (Equivalence of the two systems). $\Gamma \vdash_{\lambda} b : \beta$ if and only if $\Gamma \vdash_{\lambda_2} b : \beta$.

Proof. The proof that if $\Gamma \vdash_{\lambda_2} b : \beta$, then $\Gamma \vdash_{\lambda} b : \beta$ is by a straightforward induction on the length of the derivation of $\Gamma \vdash_{\lambda_2} b : \beta$.

The proof of the other implication is more delicate and we will show it in detail. The proof is by induction on the length of the derivation of $\Gamma \vdash_{\lambda} b : \beta$, that is, we prove that the new system is closed under the rules of the old one. We have already proved that λ_2 is closed under the variable introduction rule (see Lemma 2.5).

Using the previous lemma on closure under substitution, it is easy to prove its closure under the application rule, since, if $\Gamma \vdash_{\lambda_2} a : \alpha$ and $\Gamma \vdash_{\lambda_2} b : \alpha \rightarrow \beta$, the latter must have been formed from $\Gamma, x : \alpha \vdash_{\lambda_2} b(x) : \beta$ for some fresh variable x . Hence the judgment $\Gamma \vdash_{\lambda_2} b(a) : \beta$ is derivable as it is obtained by substituting the term a in $\Gamma \vdash_{\lambda_2} a : \alpha$ for the variable x .

Finally, suppose $\Gamma, x : \alpha \vdash_{\lambda_2} b : \beta$ is a derivable judgment. Then it must be formed from $\Gamma, x : \alpha, y_1 : \alpha_1, \dots, y_n : \alpha_n \vdash_{\lambda_2} b(y_1) \dots (y_n) : C$ by means of n abstractions, for some fresh variables y_1, \dots, y_n . Thus, by using an instance of the λ -introduction rule, we obtain that the judgment $\Gamma, x : \alpha, y_1 : \alpha_1, \dots, y_n : \alpha_n \vdash_{\lambda_2} (\lambda x : \alpha. b)(x)(y_1) \dots (y_n) : C$ is derivable, since $b \equiv b[x := x]$. Then, by $n + 1$ abstractions, we obtain a proof of the judgment $\Gamma \vdash_{\lambda_2} (\lambda x : \alpha. b) : \alpha \rightarrow \beta$, so the new system is also closed under the abstraction rule. \square

3. Extension to dependent types

The method we have described is quite elementary and only the required steps in a normalization proof are involved. Moreover, it is easy to extend it in order to consider more complex kinds of typed lambda calculi. For instance, let us consider the typed lambda calculus $\lambda^{\text{P}\times}$ obtained by using dependent types instead of simple basic types, that is, generalizing arrow types to quantified types (see, for example, Coquand (1996)), and by adding product types.

Since the rules for dependent types are not generally well known, we discuss them here in some detail, and give the equality theory in Appendix B.

Dependent types will depend on terms, hence we have to introduce contexts for the type judgments also. First, we introduce the predicate constants that we are going to use. Suppose

$$\begin{aligned} & \Gamma \vdash_{\lambda^{P\times}} \alpha_1 \text{ type} \\ & \Gamma, x_1 : \alpha_1 \vdash_{\lambda^{P\times}} \alpha_2 \text{ type} \\ & \dots \\ & \Gamma, x_1 : \alpha_1, \dots, x_{n-1} : \alpha_{n-1} \vdash_{\lambda^{P\times}} \alpha_n \text{ type.} \end{aligned}$$

Then we write $C : (x_1 : \alpha_1, \dots, x_n : \alpha_n) \text{ type } [\Gamma]$ to mean that C is a predicate constant of arity n , on the variables x_1, \dots, x_n , of type $\alpha_1, \dots, \alpha_n$, respectively, in the context Γ . In the following we will use also the notation $C : (\bar{x} : \bar{\alpha}) \text{ type } [\Gamma]$ for short. Note that we understand that the predicate constant C does not change even if the types of the variables x_1, \dots, x_n are modified by a substitution of the variables in the context Γ with suitable terms. Moreover, we assume that if $C : (\bar{x} : \bar{\alpha}) \text{ type } [\Gamma]$ is a predicate constant, and $\Gamma, x_1 : \alpha_1, \dots, x_{i-1} : \alpha_{i-1} \vdash_{\lambda^{P\times}} \alpha_i = \beta_i$ for $1 \leq i \leq n$, then $C : (\bar{x} : \bar{\beta}) \text{ type } [\Gamma]$ is a predicate constant also.

We have the following type formation rules:

$$\text{Atomic types} \quad \frac{\begin{array}{c} \Gamma \vdash_{\lambda^{P\times}} a_1 : \alpha_1 \\ \dots \\ \Gamma \vdash_{\lambda^{P\times}} a_n : \alpha_n [\bar{x}_n := \bar{a}_n] \end{array}}{C : (\bar{x} : \bar{\alpha}) \text{ type } [\Gamma]} \quad \Gamma \vdash_{\lambda^{P\times}} C(a_1, \dots, a_n) \text{ type}$$

where $\alpha_i [\bar{x}_i := \bar{a}_i]$ is shorthand for $\alpha_i [x_1 := a_1, \dots, x_{i-1} := a_{i-1}]$

$$\begin{array}{l} \text{Quantified types} \\ \text{Product types} \end{array} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} \alpha \text{ type} \quad \Gamma, x : \alpha \vdash_{\lambda^{P\times}} \beta \text{ type}}{\Gamma \vdash_{\lambda^{P\times}} (\forall x : \alpha) \beta \text{ type}} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} \alpha_1 \text{ type} \quad \dots \quad \Gamma \vdash_{\lambda^{P\times}} \alpha_n \text{ type}}{\Gamma \vdash_{\lambda^{P\times}} \alpha_1 \times \dots \times \alpha_n \text{ type}}$$

It may be useful to recall that we can define $\alpha \rightarrow \beta$ by putting $\alpha \rightarrow \beta \equiv (\forall x : \alpha) \beta$, provided that x does not appear free in β .

We will use the following rules to derive term judgments for the elements of these types:

$$\begin{array}{l} \text{Variable} \\ \text{Abstraction} \\ \text{Application} \\ \text{\textit{n}-tuple formation} \\ \text{Selection} \end{array} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} \alpha \text{ type}}{\Gamma, x : \alpha \vdash_{\lambda^{P\times}} x : \alpha} \quad \frac{\Gamma, x : \alpha \vdash_{\lambda^{P\times}} b : \beta}{\Gamma \vdash_{\lambda^{P\times}} (\lambda x : \alpha. b) : (\forall x : \alpha) \beta} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} c : (\forall x : \alpha) \beta \quad \Gamma \vdash_{\lambda^{P\times}} a : \alpha}{\Gamma \vdash_{\lambda^{P\times}} c(a) : \beta [x := a]} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} a_1 : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda^{P\times}} a_n : \alpha_n}{\Gamma \vdash_{\lambda^{P\times}} \langle a_1, \dots, a_n \rangle : \alpha_1 \times \dots \times \alpha_n} \quad \frac{\Gamma \vdash_{\lambda^{P\times}} a : \alpha_1 \times \dots \times \alpha_n}{\Gamma \vdash_{\lambda^{P\times}} a(i) : \alpha_i} \quad 1 \leq i \leq n$$

In the selection rule we have used the unusual notation $a(i)$ to indicate the i -th projection, in order to have a uniform notation for all of the elimination rules.

Since terms can appear inside dependent types, it is possible for a type to contain a redex. So, unlike the case of simply typed lambda calculus, two types can be convertible without being syntactically identical. It is then necessary to require that equal types have the same elements. Therefore we must add the following new rule:

$$\text{Conversion rule} \quad \frac{\Gamma \vdash_{\lambda^{\mathcal{P}^\times}} a : \alpha \quad \Gamma \vdash_{\lambda^{\mathcal{P}^\times}} \alpha = \beta}{\Gamma \vdash_{\lambda^{\mathcal{P}^\times}} a : \beta}$$

Of course, we have new contractions.

Definition 3.1 (β -contraction and selection contraction).

$$\beta\text{-contraction} \quad (\lambda x:\alpha.b)(a) \rightsquigarrow b[x := a]$$

$$\text{selection contraction} \quad \langle a_1, \dots, a_n \rangle (i) \rightsquigarrow a_i$$

It is important to note that, since types can depend on terms, which in general are not in normal form, there will be a normal form also for types, that is, the one that depends only on terms in normal form. Then, we have to modify our normalization algorithm in such a way that it works both on the types and on the terms we have just defined. To this end, we need a preliminary definition, which adapts Definition 2.3 to $\lambda^{\mathcal{P}^\times}$.

Definition 3.2. Let Γ be a context, T_1, \dots, T_n be a sequence of typed terms or natural numbers, α be a type and k be a natural number such that $1 \leq k \leq n$. Then, we define the k -th *argument type* of α with respect to T_1, \dots, T_n (denoted by $\alpha^{T_1, \dots, T_n; k}$) by induction on the type complexity of α , as follows:

- If $\alpha \equiv C(a_1, \dots, a_m)$, then $\alpha^{T_1, \dots, T_n; k}$ is undefined.
- If $\alpha \equiv (\forall x:\beta) \gamma$ and T_1 is a term of type β in the context Γ , then

$$\begin{aligned} \alpha^{T_1, \dots, T_n; 1} &= \beta \\ \alpha^{T_1, \dots, T_n; k+1} &= \gamma[x := T_1]^{T_2, \dots, T_n; k}, \end{aligned}$$

otherwise $\alpha^{T_1, \dots, T_n; k}$ is undefined.

- If $\alpha \equiv \alpha_1 \times \dots \times \alpha_m$ and T_1 is a natural number such that $1 \leq T_1 \leq m$, then

$$\begin{aligned} \alpha^{T_1, \dots, T_n; 1} &= \text{index} \\ \alpha^{T_1, \dots, T_n; k+1} &= \alpha_{T_1}^{T_2, \dots, T_n; k}, \end{aligned}$$

otherwise $\alpha^{T_1, \dots, T_n; k}$ is undefined.

The algorithm of reduction into normal form for types will take two arguments, that is, a context Γ and a type α , while the algorithm of reduction in normal form for terms will take three arguments, that is, a context Γ , a type α and a term e .

Conversion into normal form for types

$$\text{nf}(\alpha^\Gamma) = \begin{cases} C(\text{nf}(a_1^{\Gamma;\alpha_1}), \dots, \text{nf}(a_n^{\Gamma;\alpha_n})) & \text{if } \alpha \equiv C(a_1, \dots, a_n), \\ & C:(x_1:\beta_1, \dots, x_n:\beta_n) \text{ type } [\Gamma] \\ & \text{and } \alpha_i \equiv \beta_i[\bar{x}_i := \bar{a}_i] \\ (\forall x:\text{nf}(\beta^\Gamma)) \text{nf}(\gamma^{\Gamma,x:\beta}) & \text{if } \alpha \equiv (\forall x:\beta) \gamma \\ \text{nf}(\alpha_1^\Gamma) \times \dots \times \text{nf}(\alpha_n^\Gamma) & \text{if } \alpha \equiv \alpha_1 \times \dots \times \alpha_n \end{cases}$$

Conversion into normal form for terms

$$\text{nf}(e^{\Gamma;\alpha}) = \begin{cases} (\lambda x:\text{nf}(\beta^\Gamma).\text{nf}(e(x)^{\Gamma,x:\beta;\gamma})) & \text{if } \alpha \equiv (\forall x:\beta) \gamma \\ < \text{nf}(e(1)^{\Gamma;\alpha_1}), \dots, \text{nf}(e(n)^{\Gamma;\alpha_n}) > & \text{if } \alpha \equiv \alpha_1 \times \dots \times \alpha_n \\ x(\text{nf}(T_1^{\Gamma;\Phi_x^1}) \dots (\text{nf}(T_m^{\Gamma;\Phi_x^m}))) & \text{if } \alpha \equiv C(a_1, \dots, a_n), \\ & e \equiv x(T_1) \dots (T_m), \\ & \text{and } \Phi \equiv T_1, \dots, T_m \\ \text{nf}(c[x := a](T_1) \dots (T_m)^{\Gamma;\alpha}) & \text{if } \alpha \equiv C(a_1, \dots, a_n) \text{ and} \\ & e \equiv (\lambda x:\beta.c)(a)(T_1) \dots (T_m) \\ \text{nf}(c_i(T_1) \dots (T_m)^{\Gamma;\alpha}) & \text{if } \alpha \equiv C(a_1, \dots, a_n) \text{ and} \\ & e \equiv < c_1, \dots, c_n > (i)(T_1) \dots (T_m) \end{cases}$$

Here, the arguments T_1, \dots, T_m can be either typed terms or natural numbers indicating a projection, and we assume nf to be the identity function on natural numbers.

It is worth noting how the notion of k -th argument type, that we introduced in Definition 3.2, is used in the previous algorithm: in the case $e \equiv x(T_1) \dots (T_m)$ the recursive calls of the algorithm on the arguments T_1, \dots, T_m use the corresponding argument type of the type of the variable x , and not the types with which T_1, \dots, T_m have been derived.

It can be easily proved that the proposed normalization algorithm is partially correct, that is, if $\Gamma \vdash_{\lambda_2^{\text{P}\times}} e : \alpha$ is derivable and $\text{nf}(e^{\Gamma;\alpha})$ exists, then e and $\text{nf}(e^{\Gamma;\alpha})$ are equal terms and $\text{nf}(e^{\Gamma;\alpha})$ contains no redex (respectively, if $\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha$ type is derivable and $\text{nf}(\alpha^\Gamma)$ exists, then α and $\text{nf}(\alpha^\Gamma)$ are equal types and $\text{nf}(\alpha^\Gamma)$ contains no redex).

We should now prove the termination of the normalization algorithm on all the terms of $\lambda^{\text{P}\times}$, but the presence of the conversion rule makes it more difficult to prove normalization using our method directly. Thus, let us first recall some straightforward facts about the system $\lambda^{\text{P}\times}$ that will be useful in the following.

Lemma 3.3. The following properties are valid for the calculus $\lambda^{\text{P}\times}$:

- (i) If $\Gamma \vdash_{\lambda^{\text{P}\times}} a : \alpha$, then $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha$ type.
- (ii) If $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha = \beta$, then $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha$ type and $\Gamma \vdash_{\lambda^{\text{P}\times}} \beta$ type.
- (iii) If $\Gamma \vdash_{\lambda^{\text{P}\times}} a = b : \alpha$, then $\Gamma \vdash_{\lambda^{\text{P}\times}} a : \alpha$ and $\Gamma \vdash_{\lambda^{\text{P}\times}} b : \alpha$.

Lemma 3.4. If $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha = \beta$, then:

- (i) If $\alpha \equiv C(a_1, \dots, a_n)$ for some predicate constant $C:(\bar{x}:\bar{\alpha})$ type $[\Gamma]$, then we have $\beta \equiv C(b_1, \dots, b_n)$ for some terms b_1, \dots, b_n and, for any $1 \leq i \leq n$, $\Gamma \vdash_{\lambda^{\text{P}\times}} a_i = b_i : \alpha_i[\bar{x}_i := \bar{a}_i]$.
- (ii) If $\alpha \equiv (\forall x:\alpha_1) \alpha_2$, then $\beta \equiv (\forall x:\beta_1) \beta_2$ and $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha_1 = \beta_1$ and $\Gamma, x:\alpha_1 \vdash_{\lambda^{\text{P}\times}} \alpha_2 = \beta_2$.
- (iii) If $\alpha \equiv \alpha_1 \times \dots \times \alpha_n$, then $\beta \equiv \beta_1 \times \dots \times \beta_n$ and, for any $1 \leq i \leq n$, $\Gamma \vdash_{\lambda^{\text{P}\times}} \alpha_i = \beta_i$.

Now we can continue with our general approach: we will define a new system $\lambda_2^{\text{P}\times}$ for which the normalization algorithm always terminates. To this end, we need to introduce a notion of compatibility, which we define together with the system $\lambda_2^{\text{P}\times}$ by mutual recursion.

Definition 3.5 (First order compatibility). Given a context Γ , a type α and a finite sequence Φ whose elements are either typed terms or natural numbers, we define the notion of compatibility between α and Φ in the context Γ , and, for the case where α and Φ are compatible in the context Γ , we also associate with α and Φ the type $\beta = \text{Res}(\alpha; \Phi)$, as follows:

- α and $()$ are compatible and $\text{Res}(\alpha; ()) = \alpha$.
- If α and (T_1, \dots, T_{n-1}) are compatible and $\text{Res}(\alpha; T_1, \dots, T_{n-1}) = \gamma$, then:
 - If $\gamma \equiv C(a_1, \dots, a_m)$, then α and (T_1, \dots, T_n) are never compatible.
 - If $\gamma \equiv (\forall x:\eta) \delta$, then α and (T_1, \dots, T_n) are compatible if and only if T_n is a term such that $\Gamma \vdash_{\lambda_2^{\text{P}\times}} T_n : \eta$; in this case $\text{Res}(\alpha; T_1, \dots, T_n) = \delta[x := T_n]$.
 - If $\gamma \equiv \alpha_1 \times \dots \times \alpha_m$, then α and (T_1, \dots, T_n) are compatible if and only if T_n is a natural number such that $1 \leq T_n \leq m$; in this case, $\text{Res}(\alpha; T_1, \dots, T_n) = \alpha_{T_n}$.

We will write $(T_1, \dots, T_n) \in \text{Arg}(\alpha) [\Gamma]$ to mean that α and (T_1, \dots, T_n) are compatible in the context Γ .

The rules for type derivation for $\lambda_2^{\text{P}\times}$ will be the same as the ones of the original system; this does not imply *a priori* that the types are the same, because the terms on which we build the dependent types may be different. The rules for term derivation are as follows:

Variable introduction

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha \text{ type} \quad (T_1, \dots, T_n) \in \text{Arg}(\alpha) [\Gamma] \quad \text{Res}(\alpha; T_1, \dots, T_n) = C(b_1, \dots, b_m)}{\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} x(T_1) \dots (T_n) : C(b_1, \dots, b_m)}$$

λ -introduction

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} c : \gamma \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} a : \alpha \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} c[x := a](T_1) \dots (T_n) : C(b_1, \dots, b_m)}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} (\lambda x:\alpha.c)(a)(T_1) \dots (T_n) : C(b_1, \dots, b_m)}$$

Abstraction

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} b(x) : \beta}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} b : (\forall x:\alpha) \beta}$$

where the only free occurrence of x in $b(x)$ is the manifested one.

n -tuple introduction

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} a_i(T_1) \dots (T_n) : C(b_1, \dots, b_m) \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} a_1 : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} a_n : \alpha_n}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} \langle a_1, \dots, a_n \rangle (i)(T_1) \dots (T_n) : C(b_1, \dots, b_m)} \quad 1 \leq i \leq n$$

Product

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} a(1) : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} a(n) : \alpha_n}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} a : \alpha_1 \times \dots \times \alpha_n}$$

Finally, we add the conversion rule as in the system $\lambda^{\text{P}\times}$.

It is now possible to adapt the equivalence theorem, that is Theorem 2.7, to this new setting, since we can prove the closure of the new system under substitution in a way similar to the previous one.

Lemma 3.6 (Closure under substitution). If $\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} b : \beta$ and $\Gamma \vdash_{\lambda_2^{\text{P}\times}} a : \alpha$ are derivable judgments, then $\Gamma \vdash_{\lambda_2^{\text{P}\times}} b[x := a] : \beta[x := a]$ is a derivable judgment.

Proof. The proof is similar to the proof of Lemma 2.6, and it is only necessary to carry on the substitution along the type formation rules and the conversion rule. \square

As in the case of simply typed λ -calculus, the equivalence between the two systems follows easily from the closure under substitution of the second.

Theorem 3.7 (Equivalence of $\lambda^{\text{P}\times}$ and $\lambda_2^{\text{P}\times}$). $\Gamma \vdash_{\lambda^{\text{P}\times}} b : \beta$ if and only if $\Gamma \vdash_{\lambda_2^{\text{P}\times}} b : \beta$.

Now we need to show that the presence of the conversion rule in the system $\lambda_2^{\text{P}\times}$ does not affect the termination of the normalization algorithm; in fact this rule is the only one that is not directly suggested by the normalization algorithm. To show this result, we will prove that in $\lambda_2^{\text{P}\times}$ it is sufficient to use the conversion rule on basic types only. To this end, we need a preliminary lemma.

Lemma 3.8. If $\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} c : \gamma$ and $\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha = \beta$, then $\Gamma, x:\beta \vdash_{\lambda_2^{\text{P}\times}} c : \gamma$ with a derivation that uses only instances of the conversion rule on types whose structural complexity is less or equal to that of α .

Proof. The proof is by induction on the length of the derivation of $\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} c : \gamma$. Most of the cases follow immediately by applying the same rule to the judgement(s) obtained by the induction hypothesis. Thus, we will show here only the case when the last rule applied is an instance of the variable introduction rule for the variable x .

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha \text{ type} \quad (T_1, \dots, T_n) \in \text{Arg}(\alpha) \quad [\Gamma] \quad \text{Res}(\alpha; T_1, \dots, T_n) = C(b_1, \dots, b_m)}{\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} x(T_1) \dots (T_n) : C(b_1, \dots, b_m)}$$

Thus, $c \equiv x(T_1) \dots (T_n)$ and $\gamma \equiv C(b_1, \dots, b_m)$. By induction on n , we can prove that $(T_1, \dots, T_n) \in \text{Arg}(\beta) \quad [\Gamma]$ and $\text{Res}(\beta; T_1, \dots, T_n) = C(b_1, \dots, b_m)$. Then

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} \beta \text{ type} \quad (T_1, \dots, T_n) \in \text{Arg}(\beta) \quad [\Gamma] \quad \text{Res}(\beta; T_1, \dots, T_n) = C(b_1, \dots, b_m)}{\Gamma, x:\beta \vdash_{\lambda_2^{\text{P}\times}} x(T_1) \dots (T_n) : C(b_1, \dots, b_m)}$$

Note that in all the derivations we have built within the proof of $(T_1, \dots, T_n) \in \text{Arg}(\beta) \quad [\Gamma]$, only instances of the conversion rule on types whose complexity is lower or equal than the complexity of the type α were used. \square

The condition in the statement of the previous lemma about the use of the conversion rules only on types that are not more complex than the one in the considered equality will be essential in the proof of the next theorem.

Theorem 3.9. The following rule of *conversion on basic types*

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : C(a_1, \dots, a_n) \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} C(a_1, \dots, a_n) = C(b_1, \dots, b_n)}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : C(b_1, \dots, b_n)}$$

is sufficient to have the full conversion rule.

Proof. Suppose

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : \alpha \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha = \beta}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : \beta}$$

is the instance of the conversion rule whose admissibility we want to prove. The proof is by induction on the type complexity of the type α . If α is a basic type, we are done. Thus, let us suppose that $\alpha \equiv (\forall x:\alpha_1) \alpha_2$. Then, because of Theorem 3.4, which we can use because we have already proved the equivalence of the systems $\lambda_2^{\text{P}\times}$ and $\lambda^{\text{P}\times}$, $\beta \equiv (\forall x:\beta_1) \beta_2$ and $\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha_1 = \beta_1$ and $\Gamma, x:\alpha_1 \vdash_{\lambda_2^{\text{P}\times}} \alpha_2 = \beta_2$. But, in the system $\lambda_2^{\text{P}\times}$, the last step in any possible proof of $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : (\forall x:\alpha_1) \alpha_2$ must have been an instance of the abstraction rule whose premise is $\Gamma, x:\alpha_1 \vdash_{\lambda_2^{\text{P}\times}} c(x) : \alpha_2$. Then, since the type complexity of α_2 is lower than the type complexity of α , we get $\Gamma, x:\alpha_1 \vdash_{\lambda_2^{\text{P}\times}} c(x) : \beta_2$ by the induction hypothesis. But now, again by the induction hypothesis, we can assume that the conversion rule is valid for types less complex than α ; hence it holds for α_1 and then the previous lemma yields $\Gamma, x:\beta_1 \vdash_{\lambda_2^{\text{P}\times}} c(x) : \beta_2$. Now an application of the abstraction rule shows that $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : (\forall x:\beta_1) \beta_2$. Finally, if $\alpha \equiv \alpha_1 \times \dots \times \alpha_n$, then, by Theorem 3.4, $\beta \equiv \beta_1 \times \dots \times \beta_n$, and, for any $1 \leq i \leq n$, $\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha_i = \beta_i$. Moreover, the last step in any possible proof of $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : \alpha_1 \times \dots \times \alpha_n$ has to have been an instance of the product rule whose premises are $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c(i) : \alpha_i$ for any $1 \leq i \leq n$. But in this case we get that $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c(i) : \beta_i$, by the induction hypothesis, and hence the result follows by using an instance of the product rule. \square

We can now prove that the normalization algorithm terminates when applied to any terms of $\lambda_2^{\text{P}\times}$.

Theorem 3.10. Let $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : \alpha$. Then the computation of $\text{nf}(c^{\Gamma:\alpha})$ terminates.

Proof. The proof is by induction on the length of the derivation of $\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : \alpha$. Most of the cases are immediate since all of the rules of $\lambda_2^{\text{P}\times}$, except the conversion rule, are obtained just by reversing the order in the algorithm steps.

Only one rule, among those obtained by reversing the algorithm steps, deserves more attention, *viz.* the abstraction rule,

$$\frac{\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} b(x) : \beta}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} b : (\forall x:\alpha) \beta}$$

In this case we have that $\text{nf}(b^{\Gamma:(\forall x:\alpha)\beta})$ computes to $\lambda x:\text{nf}(\alpha^\Gamma).\text{nf}(b(x)^{\Gamma,x:\alpha;\beta})$; hence in order to show that it terminates we have to show that both $\text{nf}(\alpha^\Gamma)$ and $\text{nf}(b(x)^{\Gamma,x:\alpha;\beta})$ terminate. The termination of the latter immediately follows by the induction hypothesis, while to see that the termination of the former also follows by the induction hypothesis one has to realize that, in order for $x:\alpha$ to appear in a context, it is necessary for it to have been

introduced by a variable introduction rule (or by a weakening rule) and hence that the proof of $\Gamma \vdash_{\lambda_2^{\text{P}\times}} \alpha$ type appears somewhere within the proof of $\Gamma, x:\alpha \vdash_{\lambda_2^{\text{P}\times}} b(x) : \beta$.

The only other rule that could have been used is the conversion rule, and, after the previous theorem, we can assume that we used it in the restricted form:

$$\frac{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : C(a_1, \dots, a_n) \quad \Gamma \vdash_{\lambda_2^{\text{P}\times}} C(a_1, \dots, a_n) = C(b_1, \dots, b_n)}{\Gamma \vdash_{\lambda_2^{\text{P}\times}} c : C(b_1, \dots, b_n)}$$

Now, by the induction hypothesis, the computation of $\text{nf}(c^{\Gamma;C(a_1, \dots, a_n)})$ terminates, but then the computation of $\text{nf}(c^{\Gamma;C(b_1, \dots, b_n)})$ also terminates, since, when used on an element of a basic type, the normalization algorithm depends only on the shape of the term and the context to which it is applied and not on the type, as a direct inspection of the definition of nf shows. It is worth noting that this argument only works because we used the restricted form of the conversion rule and in the general case it is not possible to infer the termination of $\text{nf}(c^{\Gamma;\beta})$ from the termination of $\text{nf}(c^{\Gamma;\alpha})$, even if α and β are equal types, since for the non-basic types the normalization algorithm depends also on the type of the term to be normalized and not only on the term itself. \square

4. Extension to system F

We want to extend our method to a second order typed lambda calculus, namely Girard's system F (Girard 1971; Girard 1986; Girard *et al* 1989). We think that this extension is of particular interest because Girard's original proof of normalization relies on strong non-constructive logical principles. Our method allows us to give a proof in which the use of such principles is as limited as possible. To fix the notation, we recall the definition of system F in Appendix C.

In system F there are application and abstraction for both simple variables and type variables, so we have two kinds of β -contractions.

Definition 4.1 (First and second order β -contractions).

$$(\lambda x:\alpha.b)(a) \rightsquigarrow b[x := a]$$

$$(\Lambda X.b)(\alpha) \rightsquigarrow b[X := \alpha]$$

As in the previous cases, a term of the form $(\lambda x:\alpha.b)(a)$ or $(\Lambda X.b)(\alpha)$ is called a *redex* and a term is in *normal form* if it does not contain any redex.

4.1. Normalization algorithm

Also in this case we define a normalization algorithm nf that, given a context Γ , a type α and a term e of system F, provided it terminates, gives a term $\text{nf}(e^{\Gamma;\alpha})$ in normal form. As in the previous cases, since the steps of the normalization algorithm depend on types, we need a preliminary definition that adapts Definitions 2.3 and 3.2 to system F.

Definition 4.2. Let Γ be a context, T_1, \dots, T_n be a sequence of typed terms or types of system F, α be a type and k be a natural number such that $1 \leq k \leq n$. Then we define

the k -th argument type of α with respect to T_1, \dots, T_n (denoted by $\alpha^{T_1, \dots, T_n; k}$) by induction on the type complexity of α , as follows:

- If $\alpha \equiv X$, for some `TypeVar` X , then $\alpha^{T_1, \dots, T_n; k}$ is undefined.
- If $\alpha \equiv \beta \rightarrow \gamma$ and T_1 is a term of type β in the context Γ , then

$$\begin{aligned} \alpha^{T_1, \dots, T_n; 1} &= \beta \\ \alpha^{T_1, \dots, T_n; k+1} &= \gamma^{T_2, \dots, T_n; k}, \end{aligned}$$

otherwise $\alpha^{T_1, \dots, T_n; k}$ is undefined.

- If $\alpha \equiv \Pi X. \beta$ and T_1 is a type in the context Γ , then

$$\begin{aligned} \alpha^{T_1, \dots, T_n; 1} &= \text{type} \\ \alpha^{T_1, \dots, T_n; k+1} &= \beta[X := T_1]^{T_2, \dots, T_n; k}, \end{aligned}$$

otherwise $\alpha^{T_1, \dots, T_n; k}$ is undefined.

No normalization on types is required for system F, hence we will give the definition of the algorithm on typed terms only.

Algorithm of conversion into normal form

$$\text{nf}(e^{\Gamma; \alpha}) = \begin{cases} (\lambda x: \beta. \text{nf}(e(x)^{\Gamma; x; \beta; \gamma})) & \text{if } \alpha \equiv \beta \rightarrow \gamma \text{ and} \\ & x; \beta \text{ is a fresh variable} \\ (\Lambda X. \text{nf}(e(X)^{\Gamma; X \text{ TypeVar}; \beta})) & \text{if } \alpha \equiv \Pi X. \beta \\ x(\text{nf}(T_1^{\Gamma; \Phi; 1})) \dots (\text{nf}(T_n^{\Gamma; \Phi; n})) & \text{if } \alpha \text{ is a type variable,} \\ & e \equiv x(T_1) \dots (T_n) \\ & \text{and } \Phi \equiv T_1, \dots, T_n \\ \text{nf}(c[x := a](T_1) \dots (T_n)^{\Gamma; \alpha}) & \text{if } \alpha \text{ is a type variable and} \\ & e \equiv (\lambda x: \beta. c)(a)(T_1) \dots (T_n) \\ \text{nf}(c[X := \gamma](T_1) \dots (T_n)^{\Gamma; \alpha}) & \text{if } \alpha \text{ is a type variable and} \\ & e \equiv (\Lambda X. c)(\gamma)(T_1) \dots (T_n) \end{cases}$$

where T_1, \dots, T_n are typed terms or types and we assume `nf` to be the identity on types.

If the computation of $\text{nf}(e^{\Gamma; \alpha})$ terminates, it is trivial to see that the result is a term in normal form equal to e . As before, the difficulty is to prove that the algorithm `nf` always terminates when it is applied on terms of system F.

4.2. A new system to construct second order terms

Following our general approach, to prove that the normalization algorithm terminates on any term of system F, we have to define a new system F_2 on whose terms the algorithm terminates, and show that system F and F_2 are equivalent. As in the previous section, we will begin with the variable introduction rule. In this case also, the general idea is to introduce a new variable only when the resulting term is of a basic type, that is, a type variable. Hence, given a variable, we obtain a term by applying it to a sequence of arguments until we obtain an expression of basic type. In the case of system F we have two kinds of application, *viz.* to terms and to types. Since the type application can

increase the complexity of the type of the resulting term, we cannot tell in advance how many applications we need to obtain a basic type. We have already met a similar problem in the previous section, hence we use a similar solution. Once again we define a notion of compatibility by mutual recursion with the definition of the system F_2 .

Definition 4.3 (Second order compatibility). Given a context Γ , a type α and a finite sequence Φ whose elements are either typed terms or types, we define the notion of compatibility between α and Φ in the context Γ , and, when α and Φ are compatible in the context Γ , we also associate with α and Φ the type $\beta = \text{Res}(\alpha; \Phi)$, as follows:

- α and $()$ are always compatible and $\text{Res}(\alpha; ()) = \alpha$.
- If α and (T_1, \dots, T_{n-1}) are compatible and $\text{Res}(\alpha; T_1, \dots, T_{n-1}) = \gamma$, then:
 - If γ is a type variable, then α and (T_1, \dots, T_n) are never compatible.
 - If $\gamma \equiv \delta \rightarrow \eta$, then α and (T_1, \dots, T_n) are compatible if and only if T_n is a term such that $\Gamma \vdash_{F_2} T_n : \delta$; in this case $\text{Res}(\alpha; T_1, \dots, T_n) = \eta$.
 - If $\gamma \equiv \Pi X.\delta$, then α and (T_1, \dots, T_n) are compatible if and only if T_n is a type such that $\Gamma \vdash_{F_2} T_n$ type; in this case $\text{Res}(\alpha; T_1, \dots, T_n) = \delta[X := T_n]$.

We will write $(T_1, \dots, T_n) \in \text{Arg}(\alpha) [\Gamma]$ to mean that α and (T_1, \dots, T_n) are compatible in the context Γ .

Now, we can give the rules of term derivation for the system F_2 .

Variable introduction

$$\frac{\Gamma \vdash_{F_2} \alpha \text{ type} \quad (T_1, \dots, T_n) \in \text{Arg}(\alpha) [\Gamma] \quad \text{Res}(\alpha; T_1, \dots, T_n) \equiv X}{\Gamma, x:\alpha \vdash_{F_2} x(T_1) \dots (T_n) : X}$$

λ -introduction

$$\frac{\Gamma, x:\alpha \vdash_{F_2} c : \gamma \quad \Gamma \vdash_{F_2} a : \alpha \quad \Gamma \vdash_{F_2} c[x := a](T_1) \dots (T_n) : X}{\Gamma \vdash_{F_2} (\lambda x:\alpha.c)(a)(T_1) \dots (T_n) : X}$$

First order abstraction

$$\frac{\Gamma, x:\alpha \vdash_{F_2} b(x) : \beta}{\Gamma \vdash_{F_2} b : \alpha \rightarrow \beta}$$

where the only free occurrence of x in $b(x)$ is the manifested one.

Λ -introduction

$$\frac{\Gamma, Y \text{ TypeVar} \vdash_{F_2} c : \gamma \quad \Gamma \vdash_{F_2} \alpha \text{ type} \quad \Gamma \vdash_{F_2} c[Y := \alpha](T_1) \dots (T_n) : X}{\Gamma \vdash_{F_2} (\Lambda Y.c)(\alpha)(T_1) \dots (T_n) : X}$$

Second order abstraction

$$\frac{\Gamma, X \text{ TypeVar} \vdash_{F_2} b(X) : \beta}{\Gamma \vdash_{F_2} b : \Pi X.\beta}$$

where the only free occurrence of X in $b(X)$ is the manifested one.

4.3. Equivalence between F and F₂

As in the previous sections, the last step in the proof of the normalization theorem is to prove that the new system is equivalent to the old one.

There is a straightforward, but quite long proof, which uses induction on the length of the derivation of the judgement on which the substitution is performed, of the following lemma of closure under second order substitution.

Lemma 4.4 (Closure under second order substitution). F₂ is closed under second order substitution, that is, if $\Gamma \vdash_{F_2} \alpha$ type and $\Gamma, X \text{ TypeVar} \vdash_{F_2} c : \beta$ are derivable, then $\Gamma \vdash_{F_2} c[X := \alpha] : \beta[X := \alpha]$ is also derivable.

In the proof of the equivalence of F and F₂ it will be essential to be able to prove a property of a term t of F₂ using induction on the length of its derivation in F. This means not only that we must know that t is derivable in F, but also that the induction does not ‘exit’ F₂. More formally, we must prove the following lemma.

Lemma 4.5 (Embedding lemma). If $\Gamma \vdash_{F_2} c : \alpha$, then c is also a term of F and all the terms that appear in the derivation of c in F are typable in F₂. We will say that the derivation of c in F can be *embedded* in F₂.

Proof. The proof is by induction on the length of the derivation of $\Gamma \vdash_{F_2} c : \alpha$. According to the last rule used in the derivation we have:

— Variable introduction:

$$\frac{\Gamma \vdash_{F_2} \alpha \text{ type} \quad (T_1, \dots, T_n) \in \text{Arg}(\alpha) \quad [\Gamma] \quad \text{Res}(\alpha; T_1, \dots, T_n) \equiv X}{\Gamma, x:\alpha \vdash_{F_2} x(T_1) \dots (T_n) : X}$$

By the induction hypothesis, those of the T_i ’s that are terms, are also terms of F and their deduction in F can be embedded in F₂. Then we can derive $x(T_1) \dots (T_n)$ in F by n first and second order applications. Moreover, the terms that appear in the derivation of $x(T_1) \dots (T_n)$ in F are those that appear in the derivations of T_1, \dots, T_n , which are derivable in F₂ by the induction hypothesis, and $x, x(T_1), \dots, x(T_1) \dots (T_n)$. We have to prove that the latter are also derivable in F₂. By using the variable introduction rule, we have that, for $0 \leq i \leq n$, $x(T_1) \dots (T_i)(\xi_{i+1}) \dots (\xi_m) : Y$ is a term of F₂, where ξ_{i+1}, \dots, ξ_m are first and second order variables (note that $m \leq n$). Now we just need to apply first and second order abstraction rules $m - i$ times to obtain that $x(T_1) \dots (T_i)$ is a term of F₂.

— λ -introduction:

$$\frac{\Gamma, x:\alpha \vdash_{F_2} c : \gamma \quad \Gamma \vdash_{F_2} a : \alpha \quad \Gamma \vdash_{F_2} c[x := a](T_1) \dots (T_n) : X}{\Gamma \vdash_{F_2} (\lambda x:\alpha. c)(a)(T_1) \dots (T_n) : X}$$

By the induction hypothesis, $c[x := a](T_1) \dots (T_n)$ and a are terms of F and their derivation in F can be embedded in F₂. Note that the derivation of $c[x := a](T_1) \dots (T_n)$ must end with n applications, so T_1, \dots, T_n must be derivable in F, and, by the induction hypothesis, their derivations can be embedded in F₂. By the induction hypothesis, we have also that $\Gamma, x:\alpha \vdash_F c : \gamma$ and the derivation of c in F can be embedded in

F_2 . Thus, to obtain $\Gamma \vdash_F (\lambda x:\alpha.c)(a)(T_1)\dots(T_n) : X$, we can apply the first order abstraction rule one time and the first and second order application rules $n + 1$ times. We have to prove that $\lambda x:\alpha.c$ is a term of F_2 . This is done by considering the term $c[x := x](\xi_1)\dots(\xi_m) : Y$, which is deducible in F_2 , by applying first the λ -introduction rule and then $m + 1$ abstractions. As in the previous case, we can also prove that the terms $(\lambda x:\alpha.c)(a)$, $(\lambda x:\alpha.c)(a)(T_1)$, \dots , $(\lambda x:\alpha.c)(a)(T_1)\dots(T_n)$ are all deducible in F_2 .

— Λ -introduction: the proof is similar to the previous one.

— First order abstraction:

$$\frac{\Gamma, x:\alpha \vdash_{F_2} b(x) : \beta}{\Gamma \vdash_{F_2} b : \alpha \rightarrow \beta}$$

By the induction hypothesis, $b(x)$ is deducible in F and its derivation in F can be embedded in F_2 . But the derivation of $b(x)$ in F must contain a derivation of b , which is thus already embedded into F_2 .

— Second order abstraction: the proof is similar to the previous one. \square

The proof that F_2 is closed under first order substitution is the step that requires the use of non-constructive principles. To obtain this result we use part of the proof of strong normalization for system F in Krivine (1993); in particular, our *adequacy lemma* is a version for F_2 of the lemma with the same name in that book. For this reason, here we only recall the main definitions and lemmas. Let us introduce some notation:

$$\begin{aligned} \Lambda &= \bigcup_{\alpha \text{ type}} \Lambda_\alpha \text{ where } \Lambda_\alpha = \{\text{terms of system } F \text{ of type } \alpha\} \\ \Lambda^2 &= \bigcup_{\alpha \text{ type}} \Lambda_\alpha^2 \text{ where } \Lambda_\alpha^2 = \{\text{terms of } F_2 \text{ of type } \alpha\} \\ \Lambda^0 &= \bigcup_{\alpha \text{ type}} \Lambda_\alpha^0 \text{ where } \Lambda_\alpha^0 = \{\text{terms of } F_2 \text{ of type } \alpha \text{ in the} \\ &\quad \text{form } x(T_1)\dots(T_n)\}. \end{aligned}$$

Definition 4.6 (Arrow set of lambda terms of F_2). Let $\mathcal{A} \subseteq \Lambda_\alpha^2$ and $\mathcal{B} \subseteq \Lambda_\beta^2$. Then $\mathcal{A} \rightarrow \mathcal{B} = \{t \in \Lambda_{\alpha \rightarrow \beta}^2 \mid (\forall u \in \mathcal{A}) t(u) \in \mathcal{B}\}$.

Definition 4.7 (Saturated set). A set of terms $\mathcal{A} \subseteq \Lambda_\alpha^2$ is saturated if:

- $\Lambda_\alpha^0 \subseteq \mathcal{A}$.
- If $u \in \Lambda_\gamma^2$, $t \in \Lambda_\beta^2$ and $u[x := t](T_1)\dots(T_n) \in \mathcal{A}$, then $(\lambda x:\beta.u)(t)(T_1)\dots(T_n) \in \mathcal{A}$.
- If $u \in \Lambda_\gamma^2$ and $u[X := \gamma](T_1)\dots(T_n) \in \mathcal{A}$, then $(\Lambda X.u)(\gamma)(T_1)\dots(T_n) \in \mathcal{A}$.

It is easy to verify that, for every type α , Λ_α^2 is a saturated set, and that if $\mathcal{A} \subseteq \Lambda_\alpha^2$, $\mathcal{B} \subseteq \Lambda_\beta^2$ and \mathcal{B} is saturated, then $\mathcal{A} \rightarrow \mathcal{B}$ is saturated.

Definition 4.8 (Variable assignment). A variable assignment is a map from the set of type variables into saturated sets.

Let \mathcal{I} be a variable assignment, X a type variable, β a type and $\mathcal{A} \subseteq \Lambda_\beta^2$ a saturated set. Then we can define a new assignment $\mathcal{I}[X \leftarrow \mathcal{A}]$ by putting $\mathcal{I}[X \leftarrow \mathcal{A}](Y) = \mathcal{I}(Y)$ if $Y \neq X$ and $\mathcal{I}[X \leftarrow \mathcal{A}](X) = \mathcal{A}$.

Definition 4.9 (Interpretation). Let \mathcal{I} be a variable assignment. Then the interpretation $|\cdot|_{\mathcal{I}}$ is a map from types into sets of terms defined by putting, for any type α ,

- If $\alpha \equiv X$ is a type variable, then $|\alpha|_{\mathcal{I}} = \mathcal{I}(X)$.
- If $\alpha \equiv \beta \rightarrow \gamma$, then $|\alpha|_{\mathcal{I}} = |\beta|_{\mathcal{I}} \rightarrow |\gamma|_{\mathcal{I}}$.
- If $\alpha \equiv \Pi X.\beta$, then $|\alpha|_{\mathcal{I}}$ is the set of all terms t of F_2 such that $t(\gamma) \in |\beta|_{\mathcal{I}[X \leftarrow \mathcal{B}]}$ for every type γ and every saturated set $\mathcal{B} \subseteq \Lambda_{\gamma}^2$.

Note that $|\alpha|_{\mathcal{I}}$ depends only on the values that \mathcal{I} takes on the free variables of α , so if α is a closed type, then $|\alpha|_{\mathcal{I}}$ does not depend on \mathcal{I} .

Notice also that this definition is the only point in the whole method where we need to use a non-constructive logic principle, since in the third case we are defining the interpretation by an impredicative second order quantification over all saturated sets.

The following proposition states that every type is interpreted into a saturated set.

Proposition 4.10 (Interpretation correctness). Let \mathcal{I} be a variable assignment and α a type. Then $|\alpha|_{\mathcal{I}} \subseteq \Lambda_{\alpha[X_1 := \alpha_1, \dots, X_n := \alpha_n]}^2$ is a saturated set, where X_1, \dots, X_n are the free variables of α and, for $i = 1, \dots, n$, $\mathcal{I}(X_i) \subseteq \Lambda_{\alpha_i}^2$.

A standard substitution lemma holds for interpretations.

Proposition 4.11 (Substitution lemma). Let α, v be two types, X a type variable and \mathcal{I} a variable assignment, and suppose $\mathcal{A} = |v|_{\mathcal{I}}$. Then $|\alpha[X := v]|_{\mathcal{I}} = |\alpha|_{\mathcal{I}[X \leftarrow \mathcal{A}]}$.

Finally, we arrive at the main lemma.

Proposition 4.12 (Adequacy lemma). Let \mathcal{I} be a variable assignment, and u be a term of F_2 of type α with free variables $x_1 : \alpha_1, \dots, x_k : \alpha_k$, and, for $1 \leq i \leq k$, $t_i \in |\beta_i|_{\mathcal{I}}$, where $|\beta_i|_{\mathcal{I}} \subseteq \Lambda_{\alpha_i}^2$. Then $u[x_1 := t_1, \dots, x_k := t_k] \in |\beta|_{\mathcal{I}}$, for some type β such that $|\beta|_{\mathcal{I}} \subseteq \Lambda_{\alpha}^2$.

Proof. We proved in the embedding lemma that if u is a term of F_2 of type α , then it is also a term of F and its derivation in F can be embedded in F_2 . Thus, to prove this lemma we can use induction on the rules of F even if we are dealing with terms of F_2 . Once we have said that, the proof of the lemma is identical to the proof of the lemma with the same name given in Krivine (1993, page 129). \square

We are now ready to prove the main result of this section.

Theorem 4.13 (Closure of F_2 under first order substitution). F_2 is closed under first order substitution, that is, if $\Gamma, x : \gamma \vdash_{F_2} u : \alpha$ and $\Gamma \vdash_{F_2} t : \gamma$, then $\Gamma \vdash_{F_2} u[x := t] : \alpha$.

Proof. Let $x_1 : \alpha_1, \dots, x_k : \alpha_k$ be the free variables of u . If x is not one of them, then $u[x := t] \equiv u$, and the statement is trivially true.

If $x \equiv x_j$ for some $j \in \{1, \dots, k\}$, suppose X_1, \dots, X_k are type variables and define a variable assignment \mathcal{I} by putting

$$\begin{aligned} X_i &\longmapsto |X_i|_{\mathcal{I}} = \Lambda_{\alpha_i}^2 && \text{for } i \in \{1, \dots, k\} \\ Y &\longmapsto |Y|_{\mathcal{I}} = \Lambda_Y^2 && \text{if } Y \not\equiv X_i \text{ for any } i \in \{1, \dots, k\}. \end{aligned}$$

Then $x_i \in |X_i|_{\mathcal{I}}$ and $t \in |X_j|_{\mathcal{I}}$. So, by the adequacy lemma, $u[x := t] \equiv u[x_1 := x_1, \dots, x_j := t, \dots, x_k := x_k] \in \Lambda_{\alpha}^2$, and hence it is a term of F_2 . \square

As in the case of simply typed lambda calculus, once the theorem of closure under substitution is proved, the proof of the equivalence of the two systems is easy.

Theorem 4.14 (Equivalence of F and F_2). $\Gamma \vdash_F t : \alpha$ if and only if $\Gamma \vdash_{F_2} t : \alpha$.

Proof. We have already proved in the embedding lemma that if $\Gamma \vdash_{F_2} t : \alpha$, then $\Gamma \vdash_F t : \alpha$.

The other implication can be proved exactly as in the case of simply typed lambda calculus and, as there, the theorems of closure under first and second order substitution are necessary to prove that the system F_2 is closed under the first and second order application rules. \square

5. Conclusions

The method we have presented appears to be of very general applicability. Indeed, we think that it should work for any typed lambda calculus such that any type has just one introduction rule. In fact, it is only in this case that our algorithm of normalization can work. Nevertheless, we think that the proof of closure under substitution will probably become more and more demanding as the typed lambda calculus considered becomes more complex. It is, anyway, important to stress that our method makes explicit the fact that the difficult step in a proof of normalization usually corresponds to the closure under substitution of some internal interpretation (see, for example, the adequacy lemma for system F).

An interesting problem here would be to check the applicability of our method to the general case of *Pure Type Systems* (Barendregt 1992).

Moreover, as Milena Stefanova pointed out to us after reading a first draft of this paper, strong normalization follows from the weak normalization given by our algorithm. Indeed, it can be proved by induction on their derivation that all the terms of the second system are strongly normalizable.

A method of proving normalization that resembles the one we have described has been found independently and used by Ralph Matthes (see Matthes (1998, chapter 9) and Matthes and Joachimski (1998)).

Appendix A. Simply typed lambda calculus

In simply typed lambda calculus there are only basic types and function types. Hence we have the following type formation rules:

$$\begin{array}{l} \mathbf{Basic\ types} \quad \frac{C \text{ basic type}}{C \text{ type}} \\ \mathbf{Arrow\ types} \quad \frac{\alpha \text{ type} \quad \beta \text{ type}}{(\alpha \rightarrow \beta) \text{ type}} \end{array}$$

By convention, \rightarrow associates to the right, so we use $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3$ to mean $(\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_3))$.

We assume we have a countable set of variables for any type α and we use the notation $x:\alpha$ to mean that x is a variable of type α . Then, we can form lambda terms according to the following rules:

$$\begin{array}{l}
\text{Variable} \quad \frac{\alpha \text{ type}}{x:\alpha \vdash_{\lambda} x : \alpha} \\
\text{Application} \quad \frac{\Gamma \vdash_{\lambda} b : \alpha \rightarrow \beta \quad \Gamma \vdash_{\lambda} a : \alpha}{\Gamma \vdash_{\lambda} b(a) : \beta} \\
\text{Abstraction} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda} b : \beta}{\Gamma \vdash_{\lambda} (\lambda x:\alpha. b) : \alpha \rightarrow \beta}
\end{array}$$

The form of the judgments derivable by means of these rules is $\Gamma \vdash_{\lambda} t : \alpha$, that is ‘ t is a term of type α in the context Γ ’. The context Γ comprises the declarations of all the variables that appear free in the term t . We assume the standard operations of weakening, contraction and exchange between the assumptions in a context. We will write $\text{FV}(t)$ to denote the set of the free variables in t . The substitution of a term for a variable within a term is defined in the usual way (see, for example, Barendregt (1992)).

The equality relation between two lambda terms is the minimal congruence relation, with respect to λ -abstraction and application, such that the following rules hold:

$$\begin{array}{l}
\alpha\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda} b : \beta}{\Gamma \vdash_{\lambda} (\lambda y:\alpha. b[x := y]) = (\lambda x:\alpha. b) : \alpha \rightarrow \beta} \quad y \notin \text{FV}(b) \\
\eta\text{-equality} \quad \frac{\Gamma \vdash_{\lambda} b : \alpha \rightarrow \beta}{\Gamma \vdash_{\lambda} (\lambda x:\alpha. b(x)) = b : \alpha \rightarrow \beta} \quad x \notin \text{FV}(b) \\
\beta\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda} b : \beta \quad \Gamma \vdash_{\lambda} a : \alpha}{\Gamma \vdash_{\lambda} (\lambda x:\alpha. b)(a) = b[x := a] : \beta}
\end{array}$$

Appendix B. Equality for dependent types

We give here the formal definition of the two equality relations, *viz.* on types and on terms, in the lambda calculus with dependent and product types. We will not repeat the rules for type formation and term deduction that we have already shown in the main text. Since types can depend on terms, we need to define the equality also for types. The equality is defined as the minimal congruence relation, with respect to all the type and term construction operators, for which the following rules hold:

Type equality

$$\begin{array}{l}
\text{Atomic types} \quad \frac{\Gamma \vdash_{\lambda^{P \times}} a_1 = b_1 : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda^{P \times}} a_n = b_n : \alpha_n[\bar{x}_n := \bar{a}_n]}{\Gamma \vdash_{\lambda^{P \times}} C(a_1, \dots, a_n) = C(b_1, \dots, b_n)} \\
\alpha\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda^{P \times}} \beta \text{ type}}{\Gamma \vdash_{\lambda^{P \times}} (\forall x:\alpha) \beta = (\forall y:\alpha) \beta[x := y]} \quad y \notin \text{FV}(\beta)
\end{array}$$

Term equality

$$\begin{array}{l}
\alpha\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda^{P_\times}} b : \beta \quad \Gamma \vdash_{\lambda^{P_\times}} \alpha = \alpha'}{\Gamma \vdash_{\lambda^{P_\times}} (\lambda x:\alpha. b) = (\lambda y:\alpha'. b[x := y]) : (\forall x:\alpha) \beta} \quad y \notin \text{FV}(b) \\
\eta\text{-equality} \quad \frac{\Gamma \vdash_{\lambda^{P_\times}} b : (\forall x:\alpha) \beta}{\Gamma \vdash_{\lambda^{P_\times}} (\lambda x:\alpha. b(x)) = b : (\forall x:\alpha) \beta} \quad x \notin \text{FV}(b) \\
\beta\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_{\lambda^{P_\times}} b : \beta \quad \Gamma \vdash_{\lambda^{P_\times}} a : \alpha}{\Gamma \vdash_{\lambda^{P_\times}} (\lambda x:\alpha. b)(a) = b[x := a] : \beta[x := a]} \\
n\text{-tuple-equality} \quad \frac{\Gamma \vdash_{\lambda^{P_\times}} a : \alpha_1 \times \dots \times \alpha_n}{\Gamma \vdash_{\lambda^{P_\times}} a = \langle a(1), \dots, a(n) \rangle : \alpha_1 \times \dots \times \alpha_n} \\
\text{sel.-equality} \quad \frac{\Gamma \vdash_{\lambda^{P_\times}} a_1 : \alpha_1 \quad \dots \quad \Gamma \vdash_{\lambda^{P_\times}} a_n : \alpha_n}{\Gamma \vdash_{\lambda^{P_\times}} \langle a_1, \dots, a_n \rangle (i) = a_i : \alpha_i} \quad 1 \leq i \leq n
\end{array}$$

Appendix C. System F

System F is a second order typed lambda calculus, that is, there are variables for types and quantification over them is allowed. Hence the type formation rules are the following:

$$\begin{array}{l}
\text{Variable types} \quad \Gamma, X \text{ TypeVar} \vdash_F X \text{ type} \\
\text{Arrow types} \quad \frac{\Gamma \vdash_F \alpha \text{ type} \quad \Gamma \vdash_F \beta \text{ type}}{\Gamma \vdash_F \alpha \rightarrow \beta \text{ type}} \\
\text{Polymorphic types} \quad \frac{\Gamma, X \text{ TypeVar} \vdash_F \beta \text{ type}}{\Gamma \vdash_F \Pi X. \beta \text{ type}}
\end{array}$$

The rules for term derivation are similar to those of simply type lambda calculus, but there are abstraction and application for both first and second order.

$$\begin{array}{l}
\text{Variable} \quad \frac{\Gamma \vdash_F \alpha \text{ type}}{\Gamma, x:\alpha \vdash_F x : \alpha} \\
\text{First order abstraction} \quad \frac{\Gamma, x:\alpha \vdash_F b : \beta}{\Gamma \vdash_F (\lambda x:\alpha. b) : \alpha \rightarrow \beta} \\
\text{First order application} \quad \frac{\Gamma \vdash_F c : \alpha \rightarrow \beta \quad \Gamma \vdash_F a : \alpha}{\Gamma \vdash_F c(a) : \beta} \\
\text{Second order abstraction} \quad \frac{\Gamma, X \text{ TypeVar} \vdash_F b : \beta}{\Gamma \vdash_F (\Lambda X. b) : \Pi X. \beta} \\
\text{Second order application} \quad \frac{\Gamma \vdash_F c : \Pi X. \beta \quad \Gamma \vdash_F \alpha \text{ type}}{\Gamma \vdash_F c(\alpha) : \beta[X := \alpha]}
\end{array}$$

The equality between two terms of system F is the minimal congruence relation, with respect to first and second order abstraction and application, such that the following rules hold:

$$\begin{array}{l}
\alpha_1\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_F b : \beta}{\Gamma \vdash_F (\lambda y:\alpha. b[x := y]) = (\lambda x:\alpha. b) : \alpha \rightarrow \beta} \quad y \notin \text{FV}(b) \\
\eta_1\text{-equality} \quad \frac{\Gamma \vdash_F b : \alpha \rightarrow \beta}{\Gamma \vdash_F (\lambda x:\alpha. b(x)) = b : \alpha \rightarrow \beta} \quad x \notin \text{FV}(b)
\end{array}$$

$$\begin{array}{l}
\beta_1\text{-equality} \quad \frac{\Gamma, x:\alpha \vdash_F b : \beta \quad \Gamma \vdash_F a : \alpha}{\Gamma \vdash_F (\lambda x:\alpha.b)(a) = b[x := a] : \beta} \\
\alpha_2\text{-equality} \quad \frac{\Gamma, X \text{ TypeVar} \vdash_F b : \beta}{\Gamma \vdash_F (\Lambda Y.b[X := Y]) = (\Lambda X.b) : \alpha \rightarrow \beta} \quad Y \notin \text{FV}(b) \\
\eta_2\text{-equality} \quad \frac{\Gamma \vdash_F b : \Pi X.\beta}{\Gamma \vdash_F (\Lambda X.b(X)) = b : \Pi X.\beta} \quad X \notin \text{FV}(b) \\
\beta_2\text{-equality} \quad \frac{\Gamma, X \text{ TypeVar} \vdash_F b : \beta \quad \Gamma \vdash_F \alpha \text{ type}}{\Gamma \vdash_F (\Lambda X.b)(\alpha) = b[X := \alpha] : \beta[X := \alpha]}
\end{array}$$

Acknowledgments.

We would like to thank Thierry Coquand for alerting us to the problems related to the use of the conversion rule in the case of dependent types.

References

- Barendregt, H. (1984) *The Lambda Calculus, its Syntax and Semantics*, North-Holland, Amsterdam.
- Barendregt, H. (1992) Lambda Calculi with Types. In: Abramski, S., Gabbay, D. M. and Maibaum, T. S. E. (eds.) in *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.
- Coquand, T. (1996) An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming* **26** (1–3) 167–177.
- Gallier, J.H. (1990) On Girard's 'Candidats de Reductibilité'. In: Odifreddi, P. (ed.) *Logic and Computer Science*, Academic Press 123–203.
- Gandy, R. O. (1980) An early proof of normalization by A. M. Turing. In: Seldin, S.P. and Hindley, J.R. (eds.) *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, London 453–455.
- Girard, J. Y. (1971) Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: Fenstad, J. E. (ed.) *Proceedings of 2nd Scandinavian Logic Symposium*, North-Holland, Amsterdam 63–92.
- Girard, J. Y. (1986) The system F of variable types, fifteen years later. *Theoretical Computer Science* **45** 159–192.
- Girard, J. Y., Lafont, Y. and Taylor, P. (1989) *Proofs and Types*, Cambridge University Press.
- Krivine, J. L. (1993) *Lambda-Calculus, Types and Models*, Masson, Paris, Ellis Horwood, Hemel Hempstead.
- Matthes, R. (1998) *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*, Ph.D. thesis, Department of Mathematics and Informatics, University of Munich.
- Matthes, R. and Joachimski, F. (1998) Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. Submitted to the *Archive for Mathematical Logic*.
- Mayer, A. R. and Reinhold, M. B. (1986) *Type' is not a type: preliminary report*, ACM 287–295
- Valentini, S. (1994) A note on a straightforward proof of normal form theorem for simply typed λ -calculi. *Bollettino dell'Unione Matematica Italiana* **8-A** 207–213.
- Valentini, S. (1995) On the decidability of the equality theory of simply typed lambda-calculi. *Bollettino dell'Unione Matematica Italiana* **9-A** 83–93.