Electronic Notes in Theoretical Computer Science 70 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 18 pages

Eliminating Proofs from Programs

Femke van Raamsdonk [1]

Division of Mathematics and Computer Science
Faculty of Sciences, Vrije Universiteit
Amsterdam, The Netherlands
CWI, Amsterdam, The Netherlands

Paula Severi [2,3]

Dipartimento di Informatica
Università degli Studi di Torino
Torino, Italy

Abstract

This paper presents a step in the development of an operational approach to program extraction in type theory. In order to get a program from a lambda term, the logical parts need to be removed. This is done by a reduction relation $\to_\epsilon$. We study the combination of β-reduction and ϵ-reduction, both in the setting of simply typed lambda calculus and for pure type systems. In the general setting the properties confluence, subject reduction, and strong normalization are studied.

1 Introduction

A specification of a program, like for instance 'for every finite list of natural numbers there is a sorted permutation', is in general of the form $\forall x.\exists y.P(x, y)$. In type theory, this is expressed as a type $\Pi x{:}A.\Sigma y{:}B.P(x, y)$. The idea of program extraction is to extract from an inhabitant t of such a type a function $f : A \to B$ with the property that $P(x, f(x))$ holds for all x. The problem with extracting such a function f is that in general it will contain information from its correctness proof.

[1] Email: femke@cs.vu.nl
[2] Email: severi@di.unito.it
[3] The second author thanks the Vrije Universiteit and the University of Leicester for their hospitality. She is partially supported by IST-2001-322222 MIKADO; IST-2001-33477 DART.

In the last two decades several approaches to program extraction have been studied. In this paper we take as starting point the theory of specifications introduced in [19,18]. There the process of extraction a program from a proof of a specification is modeled by means of a reduction relation $\rightarrow_\sigma$. A software tool based on the theory of specifications should use the combination of β-reduction and σ-reduction. Therefore it is important to study the combination of β-reduction and σ-reduction. It turns out that this combination gives rise to several problems; for instance subject reduction does not hold.

The aim of the present paper is to study the syntactic problems caused by combining β-reduction with σ-reduction. We restrict attention to the three rules from the nine rules defining the σ-reduction relation that are the problematic ones. This subset of the σ-reduction relation is called the ϵ-reduction relation. We study the combination of β-reduction with ϵ-reduction. It is shown that when β-reduction is restricted, the properties subject reduction, confluence, and strong normalization hold for the combination of β and ϵ.

Both proofs and programs are expressed as λ-terms. However, the type of a term reveals whether it is a proof or a program. The ϵ-reduction relation $\rightarrow_\epsilon$ erases the proofs that occur in a program. Consider for example the following term:

$$(\lambda x{:}\mathsf{Nat}.\, \lambda y{:}(\mathsf{Eq}\ x\ x).\, x)\ \mathsf{zero}\ \mathsf{refl}$$

It contains a proof refl of the proposition $(\mathsf{Eq}\ x\ x)$ as an argument and also an abstraction in which y is declared to be a proof of this proposition. In order to obtain a real program the ϵ-reduction erases this proof and abstraction in two steps.

$$\begin{aligned} & (\lambda x{:}\mathsf{Nat}.\, \lambda y{:}(\mathsf{Eq}\ x\ x).\, x)\ \mathsf{zero}\ \mathsf{refl} \\ \rightarrow_\epsilon\ & (\lambda x{:}\mathsf{Nat}.\, \lambda y{:}(\mathsf{Eq}\ x\ x).\, x)\ \mathsf{zero} \\ \rightarrow_\epsilon\ & (\lambda x{:}\mathsf{Nat}.\, x)\ \mathsf{zero} \end{aligned}$$

Obviously, if we want types to be preserved under the reduction, we need to extend the epsilon reduction to types. Then the type $\Pi x{:}\mathsf{Nat}.\, (\mathsf{Eq}\ x\ x) \rightarrow \mathsf{Nat}$ reduces to $(\mathsf{Nat} \rightarrow \mathsf{Nat})$.

2 An operational approach to program extraction

The σ-reduction introduced in [19,18] describes the process of program extraction (see Figure 1). This description is independent of the type lambda calculus we choose [6].

It gives a clear understanding of the procedure and can be seen as a candidate for a system where extraction is not performed a posteriori but is integrated to the semantics of the logical system. The usual algorithms to extract programs are implementations of normalizing strategies of this reduction. Due to the addition of the σ-reduction rules, the Σ-type becomes stronger than the

Distributivity

$$\Pi x{:}U.\,(\Sigma y{:}A.P\,y) \to_\sigma \Sigma f{:}(\Pi x{:}U.\,A).\Pi x{:}U.\,P\,(f\,x)$$

$$\lambda x{:}U.\,\langle a\,,p\rangle \to_\sigma \langle \lambda x{:}U.\,a\,,\lambda x{:}U.\,p\rangle$$

$$\langle a\,,p\rangle\,u \to_\sigma \langle a\,u\,,p\,u\rangle$$

Curryfication (we assume that $x_s = \langle x_d\,,x_p\rangle$).

$$\Pi x_s{:}(\Sigma x_d{:}A.P).\,U \to_\sigma \Pi x_d{:}A.\,\Pi x_p{:}P.\,U$$

$$\lambda x_s{:}(\Sigma x_d{:}A.P).\,u \to_\sigma \lambda x_d{:}A.\,\lambda x_p{:}P.\,u$$

$$u\,\langle a\,,p\rangle \to_\sigma u\,a\,p$$

Eliminating proofs from programs

$$\Pi x_p{:}P.\,A \to_\sigma A \quad \text{if } x_p \notin FV(A)$$

$$\lambda x_p{:}P.\,a \to_\sigma a \quad \text{if } x_p \notin FV(a)$$

$$a\,p \to_\sigma a$$

Fig. 1. Definition of σ-reduction.

strong-Σ. Here we call it "ultra"- Σ [4]. Ultra-Σ has at least the expressive power of strong-Σ because the first and second projections can be coded as $\pi_1 = \lambda x_s.x_d$ and $\pi_2 = \lambda x_s.x_p$. But also it is beyond strong-Σ because it has an additional property which makes program extraction always possible. This additional property corresponds to the internalization of the notion of realizability: any specification is computationally equal to a basic specification $\Sigma x{:}A.Px$ and any proof of this specification is computationally equal to a pair $\langle a\,,p\rangle$ with $a{:}A$ and $p{:}Pa$. Then, program extraction from an inhabitant of a specification consists in just taking the first component of the pair. For instance, an inhabitant of a specification $\Pi x{:}A.\,\Sigma y{:}B.(P\,x\,y)$ reduces to a pair $\langle f\,,q\rangle$ where f is the extracted program of type $A \to B$ and q is the proof of its correctness $\Pi x{:}A.\,(P\,x\,(f\,x))$.

In order to develop an example of program extraction, we have first to illustrate how sigma reduction is extended to deal with natural numbers. In [18] we have done this extension in a general setting for the calculus of inductive constructions.

The inductive data type Nat with constructors zero and suc has the fol-

[4] In [19] a different notation is used for this type constructor.

lowing elimination rule:

$$\text{(natrec)}\ \frac{\begin{array}{l}\Gamma \vdash n : \mathsf{Nat} \\ \Gamma, x{:}\mathsf{Nat} \vdash U : s \text{ if } s \in \{\star_{\mathsf{d}}, \star_{\mathsf{p}}, \star_{\mathsf{s}}\} \\ \Gamma \vdash u_1 : U[x := n] \\ \Gamma \vdash u_2 : \Pi m{:}\mathsf{Nat}.\, (U[x := m] \to U[x := \mathsf{suc}\, m])\end{array}}{\Gamma \vdash (\mathsf{natrec}\ u_1\ u_2\ n) : U[x := n]}$$

For the sake of exposition we omitted the first argument of natrec that corresponds to the type U. The reduction for natrec is defined as follows:

$$\begin{array}{ll}(\mathsf{natrec}\ u_1\ u_2\ 0) & \to_\iota u_1 \\ (\mathsf{natrec}\ u_1\ u_2\ \mathsf{suc}\ m) & \to_\iota (u_2\, m\, (\mathsf{natrec}\ u_1\ u_2\ m))\end{array}$$

In order to obtain the correct reduction for this operator when U is a specification, we must extend the definition of σ-reduction to include the following distributivity law. In that rule we will use the following abbreviations:

$\hat{P} = \lambda n{:}Nat.\, (P\, n\, (\mathsf{natrec}\ A\ a_1\ a_2) n)$,

$\hat{p_2} = \lambda n{:}Nat.\, \lambda q{:}(\hat{P}\, n).\, (p_2\, n\, (\mathsf{natrec}\ a_1\ a_2\ n)\, q)$

(Distributivity of natrec over pairs)

$(\mathsf{natrec}\ \langle a_1\, , p_1\rangle\ \langle a_2\, , p_2\rangle\ n) \to_\sigma \langle(\mathsf{natrec}\ a_1\ a_2\ n)\, , (\mathsf{natrec}\ p_1\ \hat{p_2}\ n)\rangle$

Note that the predicate $\hat{P}$ is in fact the predicate P applied to the first component of the pair.

We are now ready to give an example of program extraction in the theory of specifications of [18] and show where it is necessary to erase the logical parts from the program. We consider the specification stating that every natural number not equal to zero has a predecessor. This specification is written in [18] as follows:

$$S = \Pi n{:}\mathsf{Nat}.\, \Sigma m{:}\mathsf{Nat}. n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$$

In first-order logic S would be written as $\forall n.\exists m.(n > 0) \to (n = \mathsf{suc}\, m)$.
We use the following abbreviations and assumptions:

- $A = \lambda n{:}\mathsf{Nat}.\, \mathsf{Nat}$,
- $P = \lambda n{:}\mathsf{Nat}.\, \lambda m{:}\mathsf{Nat}.\, n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$,
- $U = \lambda n{:}\mathsf{Nat}.\, \Sigma m{:}\mathsf{Nat}.(P\, n\, m)$,
- there are terms p_0, p_m such that $\vdash p_0 : P\, 0\, 0$ and $m : \mathsf{Nat} \vdash p_m : P\, (\mathsf{suc}\, m)\, m$,
- $q = \lambda m{:}\mathsf{Nat}.\, \lambda x{:}(U\, m).\, p_m$.

The following term is an inhabitant of the type S:

$$\begin{aligned} s = \lambda n : \mathsf{Nat}.\,(\mathsf{natrec}\ & \langle 0\,, p_0\rangle \\ & (\lambda m{:}\mathsf{Nat}.\,\lambda x{:}(U\,m).\,\langle m\,, p_m\rangle) \\ & n) \end{aligned}$$

Using the σ-reduction relation of [18], we can reduce s in two stages. In the first stage, s is reduced to a pair consisting of a program and its correctness proof:

$$\begin{aligned} s \twoheadrightarrow_\sigma\ & \langle \lambda n{:}\mathsf{Nat}.\,\mathsf{natrec}\ 0\,(\lambda m{:}\mathsf{Nat}.\,\lambda x_d{:}\mathsf{Nat}.\,\lambda x_p{:}(P\,m\,x_d).\,m)\,n, \\ & \lambda n{:}\mathsf{Nat}.\,\mathsf{natrec}\ p_0\,q\,n\rangle \end{aligned}$$

The first component of this pair contains the abstraction $\lambda x_p{:}(P\,m\,x_d).\,m$ which is expecting a proof of $(P\,m\,x_d)$ and yields a natural number m. This abstraction can be removed using one of the rules that erase logical parts from a program in the definition of σ-reduction:

$$\begin{aligned} & \lambda n{:}\mathsf{Nat}.\,\mathsf{natrec}\ 0\,(\lambda m{:}\mathsf{Nat}.\,\lambda x_d{:}\mathsf{Nat}.\,\lambda x_p{:}(P\,m\,x_d).\,m)\,n \\ & \rightarrow_\sigma\ \ (\text{by erasure, also } \rightarrow_\epsilon) \\ & \lambda n{:}\mathsf{Nat}.\,\mathsf{natrec}\ 0\,(\lambda m{:}\mathsf{Nat}.\,\lambda x_d{:}\mathsf{Nat}.\,m)\,n =_{\mathsf{def}} \mathsf{pred} \end{aligned}$$

The last term pred is the extracted program. In this paper we do not consider pairs but only the sub-reduction $\rightarrow_\epsilon$ that eliminates the logical parts from the program.

3 The simply typed case

In this section we define the calculus $\lambda^{\rightarrow}_{\beta\epsilon}$ as two copies of the simply typed λ-calculus: one for data types and one for propositions. The functions from data types to propositions are eliminated with the reduction relation $\rightarrow_\epsilon$.

The two copies of the λ-calculus are obtained by starting from a set of type variables that is partitioned in two: $\mathsf{Var}_{\mathsf{types}} = \mathsf{Var}^{\mathsf{types}}_{\mathsf{d}} \cup \mathsf{Var}^{\mathsf{types}}_{\mathsf{p}}$ and from a set of term variables that is partitioned in two: $\mathsf{Var}_{\mathsf{terms}} = \mathsf{Var}^{\mathsf{terms}}_{\mathsf{d}} \cup \mathsf{Var}^{\mathsf{terms}}_{\mathsf{p}}$.

The *types* are built in the usual way from the elements of $\mathsf{Var}_{\mathsf{types}}$ and the type constructor $\rightarrow$. Types are written as $U, V, \ldots$ The *terms* are built in the usual way from the elements of $\mathsf{Var}_{\mathsf{terms}}$, abstraction, and application. Terms are written as $u, v, \ldots$.

Now the set of types is split in a set of data types and a set of propositions, intuitively speaking by checking whether the range type is in $\mathsf{Var}^{\mathsf{types}}_{\mathsf{d}}$, the set of variables for data types, or in $\mathsf{Var}^{\mathsf{types}}_{\mathsf{p}}$, the set of variables for propositions. Similarly, the set of terms is split in a set of data terms and a set of proofs, intuitively speaking by checking whether the variable on the left spine is in $\mathsf{Var}^{\mathsf{terms}}_{\mathsf{d}}$, the set of variables for data terms, or in $\mathsf{Var}^{\mathsf{terms}}_{\mathsf{p}}$, the set of variables for proofs. For the formal definition we need the notion of heart [9] (p.116).

Definition 3.1 The *heart* of types and terms is defined inductively as follows:

$$\begin{array}{llll} \mathsf{h}(X) & = X & \mathsf{h}(x) & = x \\ \mathsf{h}(U \to V) & = \mathsf{h}(V) & \mathsf{h}(\lambda x{:}U.\, u) & = \mathsf{h}(u) \\ & & \mathsf{h}(u\, v) & = \mathsf{h}(u) \end{array}$$

Now the partitions of types and terms are as follows.

Definition 3.2

(i) The sets $\mathsf{Types_d}$ of *data types* and $\mathsf{Types_p}$ of *propositions* (or *prop-types*) are defined as follows:

$$\mathsf{Types_d} = \{U \mid \mathsf{h}(U) \in \mathsf{Var}_\mathsf{d}^\mathsf{types}\}$$
$$\mathsf{Types_p} = \{U \mid \mathsf{h}(U) \in \mathsf{Var}_\mathsf{p}^\mathsf{types}\}$$

Data types are written as $A, B, \ldots$, and propositions as $P, Q, \ldots$.

(ii) The sets $\mathsf{Terms_d}$ of *programs* (or *data-terms*) and $\mathsf{Terms_p}$ of *proofs* (or *prop-terms*) are defined as follows:

$$\mathsf{Terms_d} = \{u \mid \mathsf{h}(u) \in \mathsf{Var}_\mathsf{d}^\mathsf{terms}\}$$
$$\mathsf{Terms_p} = \{u \mid \mathsf{h}(u) \in \mathsf{Var}_\mathsf{p}^\mathsf{terms}\}$$

Programs are written as $a, b, \ldots$ and proofs as $p, q, \ldots$.

We assume that in $\lambda x{:}U.\, u$ we have that either $x \in \mathsf{Var}_\mathsf{d}^\mathsf{terms}$ and $U \in \mathsf{Types_d}$ or $x \in \mathsf{Var}_\mathsf{p}^\mathsf{terms}$ and $U \in \mathsf{Types_p}$. So abstractions are intuitively speaking well-sorted.

The ϵ-reduction relation erases propositions occurring in data-types, and proofs occurring in programs, so it is defined both on types and on terms.

Definition 3.3 The *ϵ-reduction relation on types*, notation $\to_\epsilon$, is the smallest relation that is closed under the formation of types and contains the rule:

$$(P \to A) \to_\epsilon A$$

The *ϵ-reduction relation on terms*, notation $\to_\epsilon$, is the smallest relation that is closed under the formation of terms and contains the following two rules:

$$\begin{array}{ll} \lambda x{:}P.\, a \to_\epsilon a & \text{if } x \notin FV(a) \\ a\, p \quad\;\; \to_\epsilon a & \end{array}$$

For instance, $(\lambda x{:}\bot.\, \mathsf{zero}\, x)\, p \to_\epsilon (\lambda x{:}\bot.\, \mathsf{zero}\, x) \to_\epsilon (\lambda x{:}\bot.\, \mathsf{zero}) \to_\epsilon \mathsf{zero}$, where $\bot$ is a proposition and p is a proof.

We consider a restricted β-reduction which is generated by two rewrite rules depending on whether the argument is a program or a proof.

Definition 3.4 The *β-reduction relation*, notation $\to_\beta$, is the smallest relation that is closed under the formation of terms and contains the following:

$$(\lambda x{:}A.\, u)\, a \to_\beta u[x := a]$$
$$(\lambda x{:}P.\, q)\, p \to_\beta q[x := p]$$

Note that the following terms are *no* β-redexes:

$$(\lambda x{:}A.\, u)\, p \qquad (\lambda x{:}P.\, p)\, a \qquad (\lambda x{:}P.\, a)\, p$$

The first two are no β-redexes because it is not the intention to substitute a proof for a program-variable, nor a program for a proof-variable (they are not typable in $\lambda^{\to}_{\beta\epsilon}$). Moreover, if the first term is a β-redex then confluence does not hold; see Remark 3.5. If the last term is a β-redex, then the type system does not satisfy subject reduction and strong normalization; see Remark 3.7 and Remark 3.8.

If the first two are not β-redexes, but the last one is admitted as a β-redex, then we obtain a calculus which is confluent and weakly normalizing but not strongly normalizing. Note that it is not fully extended [10]. This is the situation in the theory of specifications studied in [18].

Remark 3.5 For confluence it is essential that a term of the form $(\lambda x{:}A.\, u)\, p$ is not a β-redex. For example, we have $p \;{}_\beta\!\leftarrow (\lambda x{:}\mathsf{Nat}.\, x)\, p \to_\epsilon (\lambda x{:}\mathsf{Nat}.\, x)$. The other two non-$\beta$-redexes given above don't give rise to non-confluent reductions: terms of the form $(\lambda x{:}P.\, p)\, a$ don't give rise to more critical pairs, and terms of the form $(\lambda x{:}P.\, a)\, p$ give rise to critical pairs that are development closed (if all critical pairs are development closed then the rewriting system is confluent [22]).

The difference between the usual typing system for simply typed λ-calculus and the typing system for $\lambda^{\to}_{\beta\epsilon}$ is that here types are considered modulo ϵ-conversion. In an environment Γ we suppose that for all declarations $x{:}U$ we have that either $x \in \mathsf{Var}^{\mathsf{terms}}_{\mathsf{d}}$ and $U \in \mathsf{Types}_{\mathsf{d}}$ or $x \in \mathsf{Var}^{\mathsf{terms}}_{\mathsf{p}}$ and $U \in \mathsf{Types}_{\mathsf{p}}$. So environments are intuitively speaking well-sorted.

Definition 3.6 The typing system for $\lambda^{\to}_{\beta\epsilon}$ is defined by the following rules:

$$(\mathsf{start})\ \frac{x{:}U \in \Gamma}{\Gamma \vdash x : U} \qquad\qquad (\epsilon\text{-}\mathsf{conv})\ \frac{\Gamma \vdash u : U}{\Gamma \vdash u : U'}\ \text{ with } U =_\epsilon U'$$

$$(\mathsf{abs})\ \frac{\Gamma, x{:}U \vdash v : V}{\Gamma \vdash \lambda x{:}U.\, v : (U \to V)} \qquad (\mathsf{app})\ \frac{\Gamma \vdash v : (U \to V) \quad \Gamma \vdash u : U}{\Gamma \vdash v\, u : V}$$

For instance the term $(\lambda x{:}\bot.\, \mathsf{zero}\, x)$ is typable using the ϵ-conversion rule. This term is not typable in Coq, although functions from propositions to data types exist, like for instance `f` : `bot` $\to$ `nat` with `bot` : `Prop` and `nat` : `Set`.

Remark 3.7 For subject reduction it is essential that a term of the form $(\lambda x{:}P.\, a)\, p$ is not a β-redex. Suppose $n : \mathsf{Nat}$ and let

$$\begin{aligned} a \; &= \lambda x{:}\bot.\, (\lambda y{:}(\bot \to \bot).\, n\,(y\,x))\, x \\ a' &= \lambda x{:}\bot.\, n\,(x\,x) \end{aligned}$$

Clearly, $a \to_\beta a'$. However, a is typable and a' is not. To type a we use the equations $\mathsf{Nat} =_\epsilon (\bot \to \mathsf{Nat})$ and $((\bot \to \bot) \to \mathsf{Nat}) =_\epsilon (\bot \to \mathsf{Nat})$. The point where the proof of subject reduction (Theorem 4.13) breaks down is in the application rule. From $((\bot \to \bot) \to \mathsf{Nat}) =_\epsilon (\bot \to \mathsf{Nat})$ we cannot conclude $(\bot \to \bot) =_\epsilon \bot$.

Intuitively the problem lies on the equation $P \to \mathsf{Nat} =_\epsilon P$ which holds for all propositions P.

Remark 3.8 For strong normalization it is essential that a term of the form $(\lambda x{:}P.\, a)\, p$ is not a β-redex. As an example we reconsider the terms from Remark 3.7 but now with a different typing. We use the abbreviation $\overline{A} = A \to \bot$.

$$\begin{aligned} a \; &= \lambda x{:}\overline{\overline{\mathsf{Nat}}}.\, (\lambda y{:}(\overline{\overline{\mathsf{Nat}}} \to \bot).\, n\,(y\,x))\, x \\ a' &= \lambda x{:}\overline{\overline{\mathsf{Nat}}}.\, n\,(x\,x) \end{aligned}$$

We have $a \to_\beta a'$ and $a : \overline{\overline{\mathsf{Nat}}} \to \mathsf{Nat}$. Further we consider the following terms:

$$\begin{aligned} p \; &= \lambda x{:}\overline{\mathsf{Nat}}.\, x\,((\lambda y{:}(\overline{\mathsf{Nat}} \to \bot).\, n\,(y\,x))\, x) \\ p' &= \lambda x{:}\overline{\mathsf{Nat}}.\, x\,(n\,(x\,x)) \end{aligned}$$

We have $p \to_\beta p'$ and $p : \overline{\overline{\mathsf{Nat}}}$. Now starting from $a\,p : \mathsf{Nat}$ we have:

$$a\,p \twoheadrightarrow_\beta a'\,p' \to_\beta n\,(p'\,p') \to_\beta n\,(p'\,(n\,(p'\,p'))) \to_\beta \ldots$$

Below we list some properties of the system $\lambda^{\to}_{\beta\epsilon}$. The proofs of the first and second property are easy. The proof of the third one is outlined in the appendix for the case of pure type systems. We write $\to_{\beta\epsilon}$ for the union of $\to_\beta$ and $\to_\epsilon$.

Theorem 3.9

(i) *The reduction relation $\to_{\beta\epsilon}$ is confluent on the set of terms.*

(ii) *The reduction relation $\to_{\beta\epsilon}$ satisfies subject reduction.*

(iii) *The calculus $\lambda^{\to}_{\beta\epsilon}$ is $\beta\epsilon$-strongly normalizing.*

4 Elimination of functions in pure type systems

In this section we extend ϵ-reduction to pseudo-terms and define pure type systems with $\beta\epsilon$-conversion ($\mathrm{PTS}_{\beta\epsilon}$). The ϵ-reduction eliminates certain functions of the $\mathrm{PTS}_{\beta\epsilon}$. For the basic notions of pure type systems (PTSs) we refer to [1].

Definition 4.1 Let Var be the set of *variables*, written as $x, y, \dots$ and the set S of *sorts*, written as $s, s' \dots$. The set of *pseudo-terms* is defined by the following grammar: $\mathsf{PT} ::= \mathsf{Var} \mid \mathsf{S} \mid (\Pi\mathsf{Var}{:}\mathsf{PT}.\, \mathsf{PT}) \mid (\lambda\mathsf{Var}{:}\mathsf{PT}.\, \mathsf{PT}) \mid (\mathsf{PT}\, \mathsf{PT})$. Pseudo-terms are written as $u, v, U, V, \dots$.

The notions of heart for types and terms given in Definition 3.1 are extended to the case of PTSs as follows.

Definition 4.2 [9] (p.116) The *heart* of a pseudo-term is defined by induction: $\mathsf{h}(x) = x$, $\mathsf{h}(s) = s$, $\mathsf{h}(\Pi x{:}U.\, V) = \mathsf{h}(V)$, $\mathsf{h}(\lambda x{:}U.\, v) = \mathsf{h}(v)$ and $\mathsf{h}(u\, v) = u$.

As in the simply typed case, the notion of heart is used to partition the set of pseudo-terms into two sets: the data-pseudo-terms and the prop-pseudo-terms.

Definition 4.3 Let $\mathsf{Var_d}$ and $\mathsf{Var_p}$ be two disjoint sets such that $\mathsf{Var} = \mathsf{Var_d} \cup \mathsf{Var_p}$ and $\mathsf{S_d}$ and $\mathsf{S_p}$ be two disjoint sets such that $\mathsf{S} = \mathsf{S_d} \cup \mathsf{S_p}$. The sets $\mathsf{PT_d}$ of *data-pseudo-terms* and $\mathsf{PT_p}$ of *prop-pseudo-terms* are defined as follows:

$$\mathsf{PT_d} = \{u \mid \mathsf{h}(u) \in \mathsf{Var_d} \cup \mathsf{S_d}\}$$

$$\mathsf{PT_p} = \{u \mid \mathsf{h}(u) \in \mathsf{Var_p} \cup \mathsf{S_p}\}$$

Data-pseudo-terms are written as $a, b, A, B, \dots$ and prop-pseudo-terms are written as $p, q, P, Q, \dots$.

Similarly to the simply typed case we assume that in $\lambda x{:}U.\, u$ and $\Pi x{:}U.\, V$ we have that both x and U belong to the same set, either $\mathsf{PT_d}$ or $\mathsf{PT_p}$.

The notion of β-reduction on pseudo-terms is defined exactly as in Definition 3.4 replacing the word "terms" by "pseudo-terms".

Definition 4.4 The *ϵ-reduction relation*, notation $\to_\epsilon$, is the smallest relation that is closed under the formation of pseudo-terms and contains the rules:

$$\begin{array}{lll} \Pi x{:}P.\, A & \to_\epsilon A & \text{if } x \notin FV(a) \\ \lambda x{:}P.\, a & \to_\epsilon a & \text{if } x \notin FV(a) \\ a\, p & \to_\epsilon a & \end{array}$$

We write $\to_{\beta\epsilon}$ for the reduction relation that is the union of $\to_\beta$ and $\to_\epsilon$. Note that the sets $\mathsf{PT_d}$ and $\mathsf{PT_p}$ are closed under $\beta\epsilon$-conversion.

We assume that our specifications are well-sorted:

Definition 4.5 A specification $(\mathsf{S}, \mathsf{A}, \mathsf{R})$ of a pure type system is said to be *dual* if $\mathsf{S} = \mathsf{S_d} \cup \mathsf{S_p}$ where $\mathsf{S_d}$ and $\mathsf{S_p}$ are disjoint, $\mathsf{A} \subset \mathsf{S_d} \times \mathsf{S_d} \cup \mathsf{S_p} \times \mathsf{S_p}$ and $\mathsf{R} \subset \mathsf{S} \times \mathsf{S}$ (i.e. if $(s_1, s_2, s_3) \in \mathsf{R}$ then $s_2 = s_3$).

From now on we assume that our specifications are dual.

Pure type systems are extended with ϵ-conversion. The typing system contains judgements of the form $\Gamma \vdash u : U$ as usual. In an environment Γ

we suppose that for all declarations $x : U$ we have that both x and U belong to the same set, either $\mathsf{PT_d}$ or $\mathsf{PT_p}$. So environments are intuitively speaking well-sorted.

Definition 4.6 Let $\mathsf{S} = (\mathsf{S}, \mathsf{A}, \mathsf{R})$ be a dual specification. A *pure type system with $\beta\epsilon$-conversion* $\lambda^{\epsilon}\mathsf{S}$ ($\mathrm{PTS}_{\beta\epsilon}$) is defined by adding the ϵ-conversion rule to the rules of pure type systems:

(axiom) $\quad \emptyset \vdash s_1 : s_2 \qquad (s_1, s_2) \in \mathsf{A}$

(start) $$\frac{\Gamma \vdash U : s}{\Gamma, x : U \vdash x : U} \qquad (x \text{ fresh for } \Gamma)$$

(weakening) $$\frac{\Gamma \vdash U : s \qquad \Gamma \vdash v : V}{\Gamma, x : U \vdash v : V} \qquad (x \text{ fresh for } \Gamma)$$

(product) $$\frac{\Gamma \vdash U : s_1 \qquad \Gamma, x : U \vdash V : s_2}{\Gamma \vdash \Pi x{:}U.\, V : s_3} \qquad (s_1, s_2, s_3) \in \mathsf{R}$$

(abstraction) $$\frac{\Gamma, x : U \vdash v : V \qquad \Gamma \vdash \Pi x{:}U.\, V : s}{\Gamma \vdash \lambda x{:}U.\, v : \Pi x{:}U.\, V}$$

(application) $$\frac{\Gamma \vdash v : \Pi x{:}U.\, V \qquad \Gamma \vdash u : U}{\Gamma \vdash vu\ : V[x := u]}$$

($\beta\epsilon$-conversion) $$\frac{\Gamma \vdash u : U \quad \Gamma \vdash V : s}{\Gamma \vdash u : V} \qquad U =_{\beta\epsilon} V$$

Example 4.7

(i) Let $\mathsf{S} = (\mathsf{S}, \mathsf{A}, \mathsf{R})$ be a specification of a pure type system. We write s_{d} for the concatenation of the symbol s with d and s_{p} for the concatenation of the symbol s with p. The duplication of S is a specification $2\mathsf{S} = (2\mathsf{S}, 2\mathsf{A}, 2\mathsf{R})$ defined as follows:
   (a) $2\mathsf{S} = \{s_{\mathsf{d}}, s_{\mathsf{p}} \mid s \in \mathsf{S}\}$,
   (b) $2\mathsf{A} = \{(s_{\mathsf{d}}, s'_{\mathsf{d}}), (s_{\mathsf{p}}, s'_{\mathsf{p}}) \mid (s, s') \in \mathsf{A}\}$
   (c) $2\mathsf{R} = \{(s_{\mathsf{d}}, s'_{\mathsf{d}}, s''_{\mathsf{d}}), (s_{\mathsf{p}}, s'_{\mathsf{d}}, s''_{\mathsf{d}}), (s_{\mathsf{d}}, s'_{\mathsf{p}}, s''_{\mathsf{p}}), (s_{\mathsf{p}}, s'_{\mathsf{p}}, s''_{\mathsf{p}}) \mid (s, s', s'') \in \mathsf{R}\}$.
   The ϵ-reduction relation removes functions formed by the rules of the form $(s_{\mathsf{p}}, s'_{\mathsf{d}}, s''_{\mathsf{d}})$.

(ii) The system $\lambda^{\rightarrow}_{\beta\epsilon}$ defined in Section 3 can be viewed as a $\mathrm{PTS}_{\beta\epsilon}$ with the following specification:
   (a) $\mathsf{S} = \{\star_{\mathsf{d}}, \star_{\mathsf{p}}, \Box_{\mathsf{d}}, \Box_{\mathsf{p}}\}$,
   (b) $\mathsf{A} = \{(\star_{\mathsf{d}}, \Box_{\mathsf{d}}), (\star_{\mathsf{p}}, \Box_{\mathsf{p}})\}$,
   (c) $\mathsf{R} = \{(\star_{\mathsf{d}}, \star_{\mathsf{d}}), (\star_{\mathsf{d}}, \star_{\mathsf{p}}), (\star_{\mathsf{p}}, \star_{\mathsf{d}}), (\star_{\mathsf{p}}, \star_{\mathsf{p}})\}$.
   The ϵ-reduction removes the functions formed with the rule $(\star_{\mathsf{p}}, \star_{\mathsf{d}})$.

(iii) If we remove the sorts $\star_{\mathsf{s}}$ and $\Box_{\mathsf{s}}$ from the theory of specifications of [19],

we obtain a $\mathrm{PTS}_{\beta\epsilon}$ whose specification (MLD) is defined as follows:
(a) $\mathsf{S} = \{\star_\mathsf{d}, \star_\mathsf{p}, \Box_\mathsf{d}, \Box_\mathsf{p}\}$,
(b) $\mathsf{A} = \{(\star_\mathsf{d}, \Box_\mathsf{d}), (\star_\mathsf{p}, \Box_\mathsf{p})\}$,
(c) $\mathsf{R} = \{(\star_\mathsf{d}, \star_\mathsf{d}), (\star_\mathsf{d}, \star_\mathsf{p}), (\star_\mathsf{p}, \star_\mathsf{d}), (\star_\mathsf{p}, \star_\mathsf{p}), (\star_\mathsf{d}, \Box_\mathsf{d}), (\star_\mathsf{d}, \Box_\mathsf{p}), (\star_\mathsf{p}, \Box_\mathsf{p})\}$.
Note that this specification does not contain $(\star_\mathsf{p}, \Box_\mathsf{d})$ and hence it is not the duplication of any specification. The ϵ-reduction relation removes the functions that are formed with the rule $(\star_\mathsf{p}, \star_\mathsf{d})$.

(iv) If we remove the sorts for specifications from the theory of specifications of [18], we obtain a $\mathrm{PTS}_{\beta\epsilon}$ whose specification is defined as follows:
(a) $\mathsf{S} = \{\star^i_\mathsf{d}, \star^i_\mathsf{p} \mid i \in N\}$,
(b) $\mathsf{A} = \{(\star^i_\mathsf{d}, \star^{i+1}_\mathsf{d}), (\star^i_\mathsf{p}, \star^{i+1}_\mathsf{p}) \mid i \in N\}$
(c) $\mathsf{R} = \{(\star^i_\mathsf{d}, \star^j_\mathsf{d}), (\star^i_\mathsf{d}, \star^j_\mathsf{p}), (\star^i_\mathsf{p}, \star^j_\mathsf{d}), (\star^i_\mathsf{p}, \star^j_\mathsf{p})\}$ with $i \leq j$, or $j = 0$.
This system is the duplication of λC^∞ [17] which is a subsystem of the extended calculus of constructions [11]. The ϵ-reduction relation removes the functions that are formed rules of the form $(\star^i_\mathsf{p}, \star^j_\mathsf{d})$.

Since the rules for ϵ-reduction are orthogonal, and so is the union of the rules for β- and ϵ-reduction, we have the following result.

Theorem 4.8 (Confluence)

(i) *The reduction relation $\to_\epsilon$ is confluent on the set of pseudo-terms.*
(ii) *The reduction relation $\to_{\beta\epsilon}$ is confluent on the set of pseudo-terms.*

Since $\to_\epsilon$ is confluent and it is obviously strongly normalizing, we have that the ϵ-normal form of a pseudo-term always exists and it is unique. We denote by $\mathsf{nf}_\epsilon(u)$ the ϵ-normal form of u. The ϵ-normal form removes the prop-pseudo-terms that occur inside the data-pseudo-terms, i.e. the ϵ-normal form of a does not contain sub-terms in PT_p. In particular, it does not contain variables in Var_p.

From the typing point of view, the ϵ-reduction removes functions formed with the rules of $\mathsf{S}_\mathsf{p} \times \mathsf{S}_\mathsf{d}$. In other words, the ϵ-normal form maps typable terms in $\lambda^\epsilon\mathcal{S}$ into typable terms in a pure type system with only β-reduction and whose set of rules is obtained by subtracting $\mathsf{S}_\mathsf{p} \times \mathsf{S}_\mathsf{d}$.

Definition 4.9 Let $\mathsf{S} = (\mathsf{S}, \mathsf{A}, \mathsf{R})$ be a dual specification. We denote as S^- the specification $\mathsf{S}^- = (\mathsf{S}, \mathsf{A}, \mathsf{R}^-)$ whose set of rules is $\mathsf{R}^- = \mathsf{R} - \mathsf{S}_\mathsf{p} \times \mathsf{S}_\mathsf{d}$.

Example 4.10

(i) The verification calculus defined in [19] is the pure type system whose specification is $\mathrm{MLD}^- = \mathrm{MLD} - (\star_\mathsf{p}, \star_\mathsf{d})$ where MLD is the specification given in Example 4.7(iii).
(ii) The system $\lambda\omega_L$ defined in [16] is the duplication of $\lambda\omega$ without rules of the form $(s_\mathsf{p}, s'_\mathsf{d})$ and with the extra rule $(\star_\mathsf{d}, \Box_\mathsf{p})$ to write predicates on programs. So $\lambda\omega_L = \lambda 2\omega^- \cup (\star_\mathsf{d}, \Box_\mathsf{p})$.
(iii) The verification calculus defined in [18] is $\lambda(2C^\infty)^-$ where $\lambda 2C^\infty$ is as in

Example 4.7(iv).

The ϵ-reduction is extended to environments – without removing variables – by applying the ϵ-reduction to each declared type. The ϵ-normal form of a context is denoted as $\mathsf{nf}_\epsilon(\Gamma)$ as for pseudo-terms.

Theorem 4.11 (Elimination of functions in $\mathsf{S_p} \times \mathsf{S_d}$) *If $\Gamma \vdash u : U$ in $\lambda^\epsilon\mathsf{S}$ then $\mathsf{nf}_\epsilon(\Gamma) \vdash \mathsf{nf}_\epsilon(u) : \mathsf{nf}_\epsilon(U)$ in $\lambda\mathsf{S}^-$.*

Proof. This is proved by induction on the derivation. For the case of the $\beta\epsilon$-conversion rule, we have to prove first that if $u \to_{\beta\epsilon} u'$, then $\mathsf{nf}_\epsilon(u) \twoheadrightarrow_\beta \mathsf{nf}_\epsilon(u')$ which follows by induction on u. □

Corollary 4.12

(i) **(Preservation of $\beta\epsilon$-weak normalization)**
If the system $\lambda\mathsf{S}^-$ is β-weakly normalizing, so is $\lambda^\epsilon\mathsf{S}$.

(ii) **(Conservativity of $\lambda^\epsilon\mathsf{S}$ over $\lambda\mathsf{S}^-$)**
Let Γ, u and U legal terms in $\lambda\mathsf{S}^-$. If $\Gamma \vdash u : U$ in $\lambda^\epsilon\mathsf{S}$ then $\Gamma \vdash u : U$ in $\lambda\mathsf{S}^-$.

There is no conservativity of $\lambda^\epsilon 2S$ over λS or over $\lambda 2S$. For example $(P \to A) \to A$ is provable in $\lambda^{\to}_{\beta\epsilon}$ but it is not provable in the simply typed lambda calculus or in its duplication with only β-reduction. This type interpreted as a proposition is not even a tautology!

Theorem 4.13 (Subject reduction for $\to_\beta$) *If $\Gamma \vdash u : U$ and $u \to_\beta u'$, then $\Gamma \vdash u' : U$.*

The substitution lemma is stated as usual [1] and used in the following proof.

Proof. This is proved by induction on the derivation together with a similar statement for environments as usual. In the case of the application we use substitution lemma and the following observation:

(i) If $(\Pi x{:}A.\, V) =_{\beta\epsilon} (\Pi x{:}A'.\, V')$, then $A =_{\beta\epsilon} A'$ and $V =_{\beta\epsilon} V'$.

(ii) If $(\Pi x{:}P.\, Q) =_{\beta\epsilon} (\Pi x{:}P'.\, Q')$, then $P =_{\beta\epsilon} P'$ and $Q =_{\beta\epsilon} Q'$.

This observation does not hold for products of the form $(\Pi x{:}P.\, A)$ (see Remark 3.7). □

The problem with proving subject reduction for ϵ-reduction is that when it is attempted in the usual way, we fall into circularities: for subject reduction, we need strengthening for prop-variables, and vice versa. The trick is to find a way to prove them simultaneously.[5]

[5] Our system is not an instance of an algebraic type system [2] and hence we cannot apply that general result of subject reduction to our case.

Theorem 4.14 (Strengthening and subject reduction for $\to_\epsilon$) *We consider a singly sorted specification* S.

(i) *Let* $x \notin fv(\Delta) \cup fv(u) \cup fv(U)$.
If $\Gamma, x\!:\!P, \Delta \vdash u : U$ *then* $\Gamma, \Delta \vdash u : U$.

(ii) *If* $\Gamma \vdash u : U$ *and* $u \twoheadrightarrow_\epsilon u'$ *then* $\Gamma \vdash u' : U$.

Proof. In order to prove these two statements simultaneously, we extend the ϵ-reduction to pairs composed by environments and pseudo-terms.

$$\begin{array}{lll} [\Gamma \vdash u] & \to_\epsilon [\Gamma \vdash u'] & \text{if } u \to_\epsilon u' \\ [\Gamma \vdash u] & \to_\epsilon [\Gamma' \vdash u] & \text{if } \Gamma \to_\epsilon \Gamma' \\ [\Gamma, x{:}P, \Delta \vdash u] & \to_\epsilon [\Gamma, \Delta \vdash u] & \text{if } x \notin fv(\Delta) \cup fv(u) \end{array}$$

Then we prove subject reduction for the extended $\to_\epsilon$ dividing in two cases:

(i) Data-pseudo-terms.
If $\Gamma \vdash a : A$ and $(\Gamma, a) \twoheadrightarrow_\epsilon (\Gamma', a')$ then $\Gamma' \vdash a' : \mathsf{nf}_\epsilon(A)$.

(ii) Prop-pseudo-terms.
If $\Gamma \vdash p : P$ and $(\Gamma, p) \twoheadrightarrow_\epsilon (\Gamma', p')$ then $\Gamma' \vdash p' : P'$ with $P' =_{\beta\epsilon} P$.

These two statements are then proved by induction on the derivation. In the cases of the product and the application rules we use the following observation:

(i) If $p \twoheadrightarrow_{\beta\epsilon} s$ then $p \twoheadrightarrow_\beta s$.

(ii) If $P \twoheadrightarrow_{\beta\epsilon} \Pi x{:}U.\, Q$ then $P \twoheadrightarrow_\beta \Pi x{:}U.\, Q$.

This observation is similar to the Key Lemma in [8] (Lemma 3.4) used to prove strengthening and subject reduction for η-reduction. In our case the fact that this lemma does not hold for data-pseudo-terms complicates matters. This forces us to divide the statement of the theorem in two cases for data-pseudo-terms and prop-pseudo-terms. Since the ϵ-reduction can destroy the shape of the data-pseudo-term A we use an invariant under ϵ-reduction which is obviously $\mathsf{nf}_\epsilon(A)$. □

To prove strong normalization we can code the ϵ-redexes $(\Pi x{:}P.\, A)$ and $(\lambda x{:}P.\, a)$ as β-redexes, assuming the existence of an inhabitant for P.

Theorem 4.15 (Strong Normalization) *The duplication of the systems of the λ-cube are $\beta\epsilon$-strongly normalizing.*

Proof. It follows from Theorem A.3. In particular $\lambda^{\to}_{\beta\epsilon}$ of Example 4.7(ii) and MLD of Example 4.7(iii) are $\beta\epsilon$-strongly normalizing. □

5 Related work

In this section we compare the theory of specifications of [19] with other approaches to program extraction in type theory.

In Coq until version 6.3 [5] the extraction procedure has been performed by means of an external function based on realizability interpretations [15,14]. The σ-reduction internalizes this function. The normal form of σ-reduction corresponds to the functions $\mathcal{E}$ and $\mathcal{R}$ of [15,14] which compute the extracted program and the proof of its correctness respectively. The σ-reduction extends $\mathcal{E}$ and $\mathcal{R}$ to type systems beyond $F\omega$ and CC [18]. The definition of σ-reduction introduces new intermediate steps in the process of program extraction. An inhabitant s of a specification needs several steps of σ-reduction to reach the normal form $\langle \mathcal{E}(s)\, , \mathcal{R}(s)\rangle$. Amongst these new intermediate steps we have to erase the logical parts from the program. In [14] and [19] data types, propositions and specifications are distinguished objects. However in Coq these objects are identified at the level of the universes. We do not know how to identify objects at this level for σ-reduction without making a mess out of it. The extraction procedure in Coq version 7 [20] is experimental and also based on a reduction relation. It is difficult to compare this new approach with σ-reduction because there is almost no documentation on this new feature of Coq. Anyway in all versions of Coq so far the extracted program is given in ML and β-reduction is performed only on σ-normal forms. For the theory of specifications, however, the idea is different. A software tool based on the theory of specifications should unify the programming language and the logic. The σ-reduction should be integrated into the system as part of its evaluation mechanism. Hence it is natural to wonder, as we do in the current paper, if it is safe to combine β and σ-reductions.

In LEGO [12,21] a specification mechanism is implemented based on the notion of deliverables [3,13]. In [3,13] a category is considered whose objects are specifications and morphisms are so called first order deliverables. The specifications and their morphisms are written in ECC [11] using Σ-types. As mentioned before, the σ-reduction adds some specific properties – given in Figure 1 – to the Σ-type that makes program extraction always possible. Moreover the theory of specifications of [19] does not need to distinguish between first and second order deliverables as in [3]. The type $\Pi x{:}S.\, T$ is used to express the function space between the specifications S and T where T may depend on S. First order deliverables do not allow this dependency and hence the notion of second order deliverables has to be introduced. The types of first and second order deliverables are σ-normal forms of some particular functions between specifications. When T does not depend on x, the σ-normal form of $(S \to T)$ gives the type of first order deliverables. Similarly forgetting about some dependencies the σ-normal form of $\Pi x{:}S.\,(\Pi y{:}T.\, U)$ gives the type of second order deliverables.

The theory of specifications of [19] agrees with the main ideas behind $\lambda\omega_L$ [16]: programs and their correctness proofs are distinct objects which can be constructed in parallel. In $\lambda\omega_L$ specifications are always pairs (σ-normal forms) which get manipulated at the meta-level. The typing system consists of derivation rules given in pairs – called coupled derivation rules –

in order to construct both components simultaneously. For each constructor a couple of derivation rules is given: one for the program-part and one for the proof-part. This notion of pair of rules or coupled derivation rules is at the meta-level. In [19,18] the notions of specification and pair of [16] is made explicit in the syntax. A pair of rules of $\lambda\omega_L$ is coded as only one rule between specifications in the theory of specifications. The sigma reduction relation captures the properties of the relation between each constructor and the pair. The σ-normal form applied to the premises and conclusions of a rule in the theory of specification of [19] gives a rule between pairs that codes the pair of rules (the coupled derivation rule) in $\lambda\omega_L$.

In [4] realizability is expressed by means of a judgement relation. Though this judgement internalizes realizability in the system, the type of a realizer is still defined by an external function $\mathcal{T}$. A rule called "type extraction" which states that any realizer is typable has to be added to obtain a system where provability and Kreisel's modified realizability coincide. This rule connects the judgement of realizability with the external function $\mathcal{T}$. In the theory of specifications of [19] the type extraction rule is derivable and expressed as: if $s : S$ then $s \twoheadrightarrow_\sigma \langle a\,, p\rangle$ and $S \twoheadrightarrow_\sigma \Sigma x{:}A.P$ and hence the realizer a of P has type A.

In the theory of specifications both axioms of choice (AC) and independence of premises (IP) are derivable. AC is trivially deduced since it is the first σ-reduction rule in Figure 1. The axiom IP can also be deduced since we can prove that $(P \to \Sigma x{:}A.Q) \twoheadrightarrow_\sigma \Sigma x{:}A.(P \to Q)$ using the first rules of the first and third groups in Figure 1. The axiom of IP is not derivable in Martin Löf's type theory but it is (Kreisel modified) realizable for any Harrop formula P. In the theory of specifications it is not necessary to assume that P is Harrop (or "self-realized" or "inhabited") because the proposition P can always be erased. This shows that the theory of specifications of [19] is not conservative over Martin Löf's type theory. This observation is not surprising. Conservativity holds but over a system which does not contain data types depending on proofs.

6 Conclusions

Evaluating a program – i.e. applying β-reduction – to a program that contains logical parts is rather inefficient. In this paper we have shown that the problem of evaluating a program with logical parts is not only an efficiency issue. We may also loose some properties like subject reduction – Remark 3.7 – and strong normalization – Remark 3.8. We provided a solution to circumvent this problem which consisted of restricting β-reduction. In [7] the solution of restricting β-reduction is made stronger by requiring a term to be in σ-normal form. In this way subject reduction and strong normalization can be proved for the whole theory of specifications.

References

[1] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 1992.

[2] G. Barthe and P.-A. Melliès. On the subject reduction property for algebraic type systems. In *Proceedings of CSL '96*, number 1258 in LNCS, pages 34–57. Springer, 1997.

[3] R. Burstall and J. McKinna. Deliverables: An approach to program development in the calculus of constructions. In *Proceedings of the First Workshop on Logical Frameworks*, pages 113–121, 1990.

[4] T. Crolard. A type theory which is complete for Kreisel's modified realizability. *ENTCS*, 23(1), 1999. In the proceedings of a tutorial workshop on realizability semantics and applications, 1999.

[5] B. Barras et al. The Coq proof assistant reference manual. Technical report, INRIA, 1999.

[6] M. Fernández, I. Mackie, P. Severi and N. Szasz. A uniform approach to program extraction: Pure Type Systems with ultra-Σ types. Preprint.

[7] M. Fernández and P. Severi. An operational approach to program extraction in the calculus of constructions. To appear in *Proceedings of LOPSTR'02*, 2002.

[8] J.H. Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed lambda calculi. In *Proceedings of LICS '92*, pages 453–460. IEEE Computer Society Press, 1992.

[9] J.H. Geuvers. *Logics and Type Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[10] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In H. Ganzinger, editor, *Proceedings of RTA '96*, number 1103 in LNCS, pages 138–152, 1996.

[11] Z. Luo. ECC, an extended calculus of constructions. In *Proceedings of LICS '89*, pages 386–395. IEEE Computer Society Press, 1989.

[12] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical report, University of Edinburgh, 1992.

[13] J.H. McKinna. *Deliverables: a categorical approach to program development in type theory*. PhD thesis, University of Edinburgh, 1992.

[14] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Proceedings of POPL '89*, pages 89–104, 1989.

[15] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, 1989.

[16] E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1994.

[17] P. Severi and E. Poll. Pure type systems with definitions. In *Proceedings of LFCS '94*, number 813 in LNCS, pages 316–328. Springer, 1994.

[18] P. Severi and N. Szasz. Internal program extraction in the calculus of inductive constructions. In *Proceedings of Argentine Workshop in Theoretical Computer Science (WAIT'02)*, 2002.

[19] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1):61–87, 2001.

[20] Coq Development Team. The Coq proof assistant reference manual version 7.1 2001. URL: `http://pauillac.inria.fr/coq/doc/main.html`.

[21] LEGO Team. The LEGO library, version 1.3, 1998. URL: `http://www.dcs.ed.ac.uk/home/lego/html/release.html`.

[22] V. van Oostrom. Development closed critical pairs. In *Proceedings of HOA '95*, number 1074 in LNCS, pages 185–200. Springer, 1995.

A Strong normalization

In this appendix we outline the proof of Theorem 4.15. Though a semantic proof of $\beta\epsilon$-strong normalization for the duplications of the systems of the λ-cube is possible, we prefer to do a more general syntactic proof which consists in coding the ϵ-redexes as β-redexes. To code $(\Pi x{:}P.\, A)$ and $(\lambda x{:}P.\, a)$ as β-redexes, we suppose the existence of a function $\mathsf{inh}_\Gamma : \mathsf{Ctx} \times \mathsf{PT} \to \mathsf{PT}$ such that $\mathsf{inh}_\Gamma(P)$ gives an inhabitant of P in Γ.

Definition A.1 Let $\mathsf{inh} : \mathsf{Ctx} \times \mathsf{PT} \to \mathsf{PT}$ and $\Gamma \vdash u : U$ or $u = s$. Then $[\![u]\!]_\Gamma^{\mathsf{inh}}$ (we omit the subscript and write $[\![u]\!]_\Gamma$) is defined inductively by the clauses shown in Figure A.1.

Note that $[\![u]\!]_\Gamma = u$ if u is in ϵ-normal form.

Lemma A.2 *If* $u \to_{\beta\epsilon} u'$ *then* $[\![u]\!]_\Gamma \to_\beta^+ [\![u']\!]_\Gamma$

This lemma is proved by induction on the structure of the term u.

Theorem A.3 *Let* $\mathsf{S}' = (\mathsf{S}', \mathsf{A}', \mathsf{R}') \supset \mathsf{S} = (\mathsf{S}, \mathsf{A}, \mathsf{R})$. *Suppose the following conditions are satisfied.*

(i) *We have that* $\lambda\mathsf{S}'$ *contains enough functions to code the* ϵ*-redexes of* $\lambda^\epsilon\mathsf{S}$*: for all* $(s_1, s_2) \in \mathsf{R}$ *there is* $s_3 \in \mathsf{S}'$ *such that* $(s_2, s_3) \in \mathsf{A}'$ *and* $(s_1, s_3) \in \mathsf{R}'$.

(ii) $\lambda S'$ *is weakly* β*-normalizing, and satisfies unicity of types (up to* β*-conversion).*

(iii) inh *is a function that yields enough inhabitants of* λS*, i.e. for all* U *such that* $\Gamma \vdash U : s$ *we have that* $\Gamma \vdash \mathsf{inh}_\Gamma(U) : U$ *and this function is invariant under* β*-reduction and thinning of environments.*

$$[\![s]\!]_\Gamma = s$$

$$[\![x]\!]_\Gamma = x$$

$$[\![\Pi x{:}U.\,V]\!]_\Gamma = \begin{cases} (\lambda x{:}[\![U]\!]_\Gamma.\,[\![V]\!]_{\Gamma,x:U})\,\mathsf{inh}_{\mathsf{nf}_\epsilon(\Gamma)}(\mathsf{nf}_\epsilon(U)) \\ \qquad \text{if } U \in \mathsf{PT_p} \text{ and } V \in \mathsf{PT_d} \\ (\Pi x{:}[\![U]\!]_\Gamma.\,[\![V]\!]_{\Gamma,x:U}) \\ \qquad \text{otherwise} \end{cases}$$

$$[\![\lambda x{:}U.\,u]\!]_\Gamma = \begin{cases} (\lambda x{:}[\![U]\!]_\Gamma.\,[\![u]\!]_{\Gamma,x:U})\,\mathsf{inh}_{\mathsf{nf}_\epsilon(\Gamma)}(\mathsf{nf}_\epsilon(U)) \\ \qquad \text{if } U \in \mathsf{PT_p} \text{ and } u \in \mathsf{PT_d} \\ (\lambda x{:}[\![U]\!]_\Gamma.\,[\![u]\!]_{\Gamma,x:U}) \\ \qquad \text{otherwise} \end{cases}$$

$$[\![v\,u]\!]_\Gamma = \begin{cases} (\lambda x{:}\mathsf{nf}_{\beta\epsilon}(P).\,[\![v]\!]_\Gamma)\,[\![u]\!]_\Gamma \\ \qquad \text{if } \Gamma \vdash v : \Pi x{:}P.\,A \text{ and } \Gamma \vdash u : P \\ [\![v]\!]_\Gamma\,[\![u]\!]_\Gamma \\ \qquad \text{otherwise} \end{cases}$$

Fig. A.1. Codification of ϵ-redexes.

Then the following statements hold:

(i) *If $\Gamma \vdash u : U$ in $\lambda^\epsilon S$ then $[\![\Gamma]\!] \vdash [\![u]\!]_\Gamma : [\![U]\!]_\Gamma$ in $\lambda S'$.*

(ii) *If $\lambda S'$ is β-strongly normalizing then $\lambda^\epsilon S$ is $\beta\epsilon$-strongly normalizing.*

Proof. Note that by the second hypothesis and Corollary 4.12, $\lambda^\epsilon S$ is $\beta\epsilon$-weakly normalizing and satisfies unicity of types (up to $\beta\epsilon$-conversion).

(i) It is proved by induction on the derivation. In the case of the product rule, we need to use the first hypothesis. To type the β-redex that codes the product $\Pi x{:}P.\,A$ we need to use a rule which may not be available in S but in S'.

(ii) Using the previous part.

□