A Uniform Approach to Program Extraction: Pure Type Systems with Ultra Σ-types [*]

Maribel Fernández and Ian Mackie
Department of Computer Science, King's College London,
Strand, WC2R 2LS London, U.K.
{ian,maribel}@dcs.kcl.ac.uk

and

Paula Severi
Departimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
severi@di.unito.it

and

Nora Szasz
In.Co., Facultad de Ingeniería,
Herrera y Reissig 525, Montevideo, Uruguay
nora@fing.edu.uy

Abstract

We introduce Ultra Σ-types in Pure Type Systems generalizing earlier work on program extraction in Type Theory. This general and comprehensive setting includes the work by Paulin based on realizability interpretations, Burstall and McKinna's Theory of Specifications and Deliverables, Poll's Programming Logic, Severi and Szasz's Theory of Specifications. We show how to express all these theories as particular Pure Type Systems with Ultra Σ-types. This general presentation helps understanding and comparing all these different approaches to program extraction.
Keywords: Specifications, program extraction, typed lambda calculus, pure type systems, Σ-types.

Resumen

Se presenta una extension de los Sistemas de Tipos Puros con tipos Ultra-Σ, que generaliza los trabajos previos sobre extracción de programas en Teoría de Tipos. Este marco general incluye los trabajos de Paulin basados en interpretaciones de realización, la Teoría de Especificaciones y Librables de Burstall y McKinna, la Lógica de Programación de Poll, y la Teoría de Especificaciones de Severi y Szasz. Se muestra como expresar todas estas teorías como casos particulares de Sistemas de Tipos Puros con tipos Ultra-Σ. Esta presentación general permite una mejor comprensión y comparación de los diferentes enfoques existentes de la extracción de programas.
Palabras claves: Especificaciones, extracción de programas, cálculo lambda tipado, sistemas de tipos puros, tipos Σ.

[*]This work has been partially supported by the ECOS-Sud program of cooperation between France and Uruguay. The third author is also partially supported by IST-2001-322222 MIKADO; IST-2001-33477 DART.

1 Introduction

A specification of a program, such as *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 $f : A \to B$ such that $P(x, f(x))$ holds for all x. The problem with standard extraction methods is that in general the extracted program contains superfluous information from its correctness proof.

In the last two decades several approaches to program extraction in Type Theory have been studied. In Coq [1] the extraction procedure is performed by means of an external function based on realizability interpretations [11, 10]. Nuprl's term extraction is also implemented by an external function, which uses extract forms associated to proof rules to build a term (see [3]). In the Theory of Specifications and Deliverables [2] a specification is a pair consisting of a data type and a predicate over it. Σ-types are used to put together both the components of specifications and the functions between them. In the Programming Logic [12] Σ-types are not used to write specifications instead the notion of coupled derivation rules is introduced. In the Theory of Specifications (introduced as an extension of Martin Löf's Type Theory in [14] and adapted to the Calculus of Constructions in [15]) pairs are part of the syntax, and the process of extracting a program from a proof of its specification is modelled by means of a reduction relation $\to_\sigma$ which erases all superfluous information from the program.

In this paper we set all these theories in a common framework. Our starting point are Pure Type Systems with Σ-types. We add a reduction relation called $\to_\sigma$ which extracts a program from a proof of its specification and erases all superfluous information from the program. Due to the addition of the σ-reduction rules Σ-types become stronger than the strong-Σ [6], we call them Ultra Σ. Ultra Σ has at least the power of strong-Σ since it is possible to code the first and second projections (it is not necessary to add them as primitives). Moreover, Ultra Σ 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 $\langle f\,, q\rangle$ where $f{:}\,A \to B$ is the extracted program and q is the proof of its correctness $\Pi x{:}A.(P\,x\,(f\,x))$.

We give a second presentation of Pure Type Systems with Σ-types where we use the same notation of pairs for types and objects, i.e. we write $\langle A\,, \lambda x{:}A.P\rangle$ instead of $\Sigma x{:}A.P$. The advantage of this presentation is that the σ-reduction works uniformly with objects and types.

The operational description of program extraction using σ-reduction is independent of the particular Type Theory specified in the Pure Type System. We show how the existing approaches to program extraction in Type Theory can all be modelled in our framework: the Theory of Specifications based on Martin Löf's Type Theory [14], the Theory of Specifications based on the Calculus of Constructions [15, 5], the method of extracting programs based on realizability interpretations by C. Paulin [10] and the work of Crolard [4], the Theory of Specifications and Deliverables [2] and the method of program development by data refinement [7], Poll's $\lambda\omega_L$ [12]. In this sense, extended Pure Type Systems with σ-reduction offer a uniform approach to program extraction in Type Theory.

We finish the paper with some examples of other applications of this formalism, such as the derivation of logical axioms (Axiom of Choice and Independence of Premisses).

The rest of the paper is organized as follows. We define Pure Type Systems with Σ-types in Section 2. Ultra Σ-types and Pairs are introduced in Section 3. Section 4 gives several examples of systems that can be seen as particular cases of our framework, and Section 5 shows the derivation of logical axioms.

Axiom

$$\vdash k_1{:}k_2 \quad (k_1,k_2)\in\mathcal{A}$$

Start

$$\frac{\Gamma\vdash U{:}k}{\Gamma, x{:}U\vdash x{:}U}\; x\ \Gamma\text{-}\mathit{fresh}$$

Weakening

$$\frac{\Gamma\vdash U{:}k \quad \Gamma\vdash v{:}V}{\Gamma, x{:}U\vdash v{:}V}\; x\ \Gamma\text{-}\mathit{fresh}$$

β-Conversion

$$\frac{\Gamma\vdash u{:}U \quad \Gamma\vdash U'{:}k}{\Gamma\vdash u{:}U'}\; U{=}_\beta U'$$

Product $\quad (k_1,k_2,k_3)\in\mathcal{R}$

$$\frac{\Gamma\vdash U{:}k_1 \quad \Gamma, x{:}U\vdash V{:}k_2}{\Gamma\vdash \Pi x{:}U.V{:}k_3}$$

Abstraction

$$\frac{\Gamma, x{:}U\vdash v{:}V \quad \Gamma\vdash \Pi x{:}U.V{:}k}{\Gamma\vdash \lambda x{:}U.v{:}\Pi x{:}U.V}$$

Application

$$\frac{\Gamma\vdash v{:}\Pi x{:}U.V \quad \Gamma\vdash u{:}U}{\Gamma\vdash v\,u{:}V[u/x]}$$

Figure 1: Typing rules for Pure Type Systems

Sigma Formation $(k_1,k_2,k_3)\in\mathcal{R}'$

$$\frac{\Gamma\vdash A{:}k_1 \quad \Gamma, x{:}A\vdash P{:}k_2}{\Gamma\vdash \Sigma x{:}A.P{:}k_3}$$

Sigma Object

$$\frac{\Gamma\vdash a{:}A \quad \Gamma\vdash p{:}P[a/x] \quad \Gamma\vdash \Sigma x{:}A.P{:}k}{\Gamma\vdash \langle a\,,p\rangle{:}\Sigma x{:}A.P}$$

First projection

$$\frac{\Gamma\vdash s{:}\Sigma x{:}A.P}{\Gamma\vdash \pi_1 s{:}A}$$

Second projection

$$\frac{\Gamma\vdash s{:}\Sigma x{:}A.P}{\Gamma\vdash \pi_2 s{:}P[\pi_1 s/x]}$$

π-conversion

$$\frac{\Gamma\vdash u{:}U \quad \Gamma\vdash U'{:}k}{\Gamma\vdash u{:}U'}\quad U =_\pi U'$$

Figure 2: Typing rules for Σ-types

2 Pure Type Systems with Σ-types

We assume the reader to be familiar with the notion of Pure Type Systems, and recall the typing rules for Pure Type Systems with Σ-types.

Let $(\mathcal{S},\mathcal{A},\mathcal{R})$ be the specification of a Pure Type System, i.e. a set $\mathcal{S}$ of sorts, a set $\mathcal{A}\subseteq\mathcal{S}\times\mathcal{S}$ of axioms, and a set $\mathcal{R}\subseteq\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ of rules. We write $(k_1,k_2)\in\mathcal{R}$ when $(k_1,k_2,k_2)\in\mathcal{R}$. The set of pseudoterms for Pure Type Systems is defined as the least set that contains a set of variables, the set $\mathcal{S}$ of sorts, abstractions $\lambda x{:}U.u$, applications $(u\,v)$, product $\Pi x{:}U.V$.

Definition 2.1. A Pure Type System (PTS or PTS_β) is defined by the rules shown in Figure 1.

We extend Pure Type Systems to include Σ-types. The set of pseudoterms is extended with Σ-expressions $\Sigma x{:}U.V$, pairs $\langle u\,,v\rangle$ and constants π_1 and π_2 for projections. The reduction rules for projections are defined as usual: $\pi_1\langle a\,,p\rangle\to_\pi a$ and $\pi_2\langle a\,,p\rangle\to_\pi p$.

A specification $\mathsf{S}=(\mathcal{S},\mathcal{A},\mathcal{R},\mathcal{R}')$ for a Pure Type System extended with Σ-types consists of a set $\mathcal{S}$ of sorts, a set $\mathcal{A}\subseteq\mathcal{S}\times\mathcal{S}$ of axioms, a set $\mathcal{R}\subseteq\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ of rules for the Π-constructor and a set $\mathcal{R}'\subseteq\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ of rules for the Σ-constructor.

Definition 2.2. A *Pure Type System with Σ-types* (Σ-PTS or $\Sigma{-}\mathrm{PTS}_\beta$) is defined by adding the rules in Figure 2 to the ones of Pure Type Systems.

3 Ultra Σ-types and Pairs

In this section we make two different extensions to Pure Type Systems. First we introduce the notion of Ultra Σ-types. Pure Type Systems with Ultra Σ-types are Pure Type Systems with Σ-types extended with a new reduction relation $\to_\sigma$ and a conversion rule $=_\sigma$. The second extension is based on Pairs.

Splitting

$$x_s \quad \rightarrow_\sigma \quad \langle x_d\,, x_p \rangle \text{ if } x_s : \Sigma x{:}A.P$$

Eliminating proofs from programs

$$\begin{array}{lll} \Pi x_p{:}P.A & \rightarrow_\sigma A & \text{if } x_p, x_s \notin FV(A) \\ \lambda x_p{:}P.a & \rightarrow_\sigma a & \text{if } x_p, x_s \notin FV(a) \\ a\,p & \rightarrow_\sigma a & \end{array}$$

Curryfication

$$\begin{array}{ll} \Pi x_s{:}(\Sigma x_d{:}A.P).U & \rightarrow_\sigma \Pi x_d{:}A.\Pi x_p{:}P.U \\ \lambda x_s{:}(\Sigma x_d{:}A.P).u & \rightarrow_\sigma \lambda x_d{:}A.\lambda x_p{:}P.u \\ u\,\langle a\,, p\rangle & \rightarrow_\sigma u\,a\,p \end{array}$$

Distributivity

$$\begin{array}{ll} \Pi x{:}U.(\Sigma y{:}A.P) & \rightarrow_\sigma \Sigma f{:}(\Pi x{:}U.A).\Pi x{:}U.P[(f\,x)/y] \\ \lambda x{:}U.\langle a\,, p\rangle & \rightarrow_\sigma \langle \lambda x{:}U.a\,, \lambda x{:}U.p\rangle \\ \langle a\,, p\rangle\,u & \rightarrow_\sigma \langle a\,u\,, p\,u\rangle \end{array}$$

Figure 3: Definition of $\rightarrow_\sigma$ using Σ-types

We assume the set of variables is split in three pairwise disjoint sets: data-variables, prop-variables and spec-variables. We denote data-variables by x_d, y_d, prop-variables by x_p, y_p and spec-variables by x_s, y_s. Similarly the set of sorts is split in three pairwise disjoint sets: data-sorts, prop-sorts and spec-sorts.

The *heart of a pseudoterm* is defined as follows:

$\mathsf{heart}(u) = u$ if u is either a variable, a sort, a Σ-type or a pair.

$\mathsf{heart}(\pi_1 u) = \mathsf{heart}(\pi_2 u) = \mathsf{heart}(u)$

$\mathsf{heart}(\Pi x{:}U.V) = \mathsf{heart}(\lambda x{:}U.V) = \mathsf{heart}(V\,U) = \mathsf{heart}(V)$.

Definition 3.1. A pseudoterm U is a *data-pseudoterm* if $\mathsf{heart}(U)$ is a data-variable or a data-sort. We denote data-pseudoterms by A, B, a, b ….

A pseudoterm U is a *prop-pseudoterm* if $\mathsf{heart}(U)$ is a prop-variable or a prop-sort. We denote prop-pseudoterms by P, Q, $p, q, \ldots$.

A pseudoterm U is a *spec-pseudoterm* if $\mathsf{heart}(U)$ is a spec-variable, a spec-sort, a pair or a Σ-expression. We denote spec-pseudoterms by S, T, $s, t, \ldots$.

We use the metavariables U, V, u, v … for any pseudoterm.

We make the following conventions: [1]

1. In $\lambda x{:}U.V$ or $\Pi x{:}U.V$ we have that both x and U are data-pseudoterms, both are prop-pseudoterms or both are spec-pseudoterms. The same restriction applies to any typing context Γ, i.e. if $x{:}U \in \Gamma$ then both x and U are data-pseudoterms, both are prop-pseudoterms or both are spec-pseudoterms.

2. For all pseudoterms $\langle u\,, v\rangle$, we assume that u is a data-pseudoterm and v is a prop-pseudoterm.

In Figure 3 we show the reduction rules defining $\rightarrow_\sigma$ for Ultra Σ-types and their objects. This reduction gives an operational semantics to program extraction when we use the Σ-type to express our specifications of programs. We assume that for each spec-variable $x_s : \Sigma x{:}A.P$ there is an associated pair $\langle x_d\,, x_p\rangle$. This is expressed by the rule Splitting (note that x_s is a variable of the PTS, but in the rewrite system it is treated as a constant). We denote $\twoheadrightarrow_\sigma$ the reflexive, symmetric and transitive closure of $\rightarrow_\sigma$.

The projections π_1 and π_2 can be coded for each type of the form $\Sigma x_d{:}A.P$. They are coded as $\pi_1 =^{\mathsf{def}} \lambda x_s : (\Sigma x_d{:}A.P).x_d$ and $\pi_2 =^{\mathsf{def}} \lambda x_s : (\Sigma x_d{:}A.P).x_p$.

The following notion of *completeness* is needed to restrict substitution of spec-variables $x_s : \Sigma x{:}A.P$. We also need a modified definition of *freshness* with respect to a typing context.

[1] See [14, 15] where these restrictions are imposed by a grammar.

$$\sigma\text{-conversion}\quad \frac{\Gamma\vdash u{:}U \quad \Gamma\vdash U'{:}k}{\Gamma\vdash u{:}U'}\; U{=_\sigma}U' \qquad \text{spec-variable}\quad \frac{\Gamma\vdash \langle x_d\,,x_p\rangle{:}\Sigma x_d{:}A.P \quad \Gamma\vdash \Sigma x_d{:}A.P{:}k}{\Gamma\vdash x_s{:}\Sigma x_d{:}A.P}\; x_s \text{ is not in } \Gamma$$

$$\text{data-variable}\quad \frac{\Gamma\vdash x_s{:}\Sigma x_d{:}A.P}{\Gamma\vdash x_d{:}A}\; x_d \text{ is not in } \Gamma \qquad \text{prop-variable}\quad \frac{\Gamma\vdash x_s{:}\Sigma x_d{:}A.P}{\Gamma\vdash x_p{:}P}\; x_p \text{ is not in } \Gamma$$

Figure 4: Typing Ultra Σ-types

Splitting

$x_s \to_\sigma \langle x_d\,, x_p\rangle$ if $x_s : \langle A\,, P\rangle$

Eliminating proofs from programs

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

Curryfication

$$\begin{array}{ll} \Pi x_s{:}\langle A\,, P\rangle.U & \to_\sigma \Pi x_d{:}A.\Pi x_p{:}(P\,x_d).U \\ \lambda x_s{:}\langle A\,, P\rangle.u & \to_\sigma \lambda x_d{:}A.\lambda x_p{:}(P\,x_d).u \\ u\,\langle a\,, p\rangle & \to_\sigma u\,a\,p \end{array}$$

Distributivity

$$\begin{array}{ll} \Pi x{:}U.\langle A\,, P\rangle & \to_\sigma \langle \Pi x{:}U.A\,, \lambda f{:}\Pi x{:}U.A.\Pi x{:}U.(P\,(f\,x))\rangle \\ \lambda x{:}U.\langle a\,, p\rangle & \to_\sigma \langle \lambda x{:}U.a\,, \lambda x{:}U.p\rangle \\ \langle a\,, p\rangle\,u & \to_\sigma \langle a\,u\,, p\,u\rangle \end{array}$$

Figure 5: Definition of σ-reduction using only the pair constructor

- The variable x_d (resp. x_p) is *complete* for a term u if $x_s \notin FV(u)$. The variable x_d (resp. x_p) is *fresh* for a context Γ (Γ-fresh for short) if x_s and x_d (resp. x_p) do not occur in Γ.
- The variable x_s is *complete* for a term u if $x_d, x_p \notin FV(u)$. It is *fresh* for a context Γ if x_s, x_d and x_p do not occur in Γ.

Whenever a substitution $u[v/x]$ is performed we require that the variable x is complete for u.

To avoid the problems shown in [13], β-reduction is restricted to σ-normal forms. It is defined as the least compatible relation on σ-normal forms that contains the β-rules:

$$\begin{array}{lll} (\lambda x_d{:}A.b)\,a & \to_\beta & b[a/x_d] \\ (\lambda x_p{:}P.p)\,q & \to_\beta & p[q/x_p] \\ (\lambda x_s{:}S.t)\,s & \to_\beta & t[s/x_s] \end{array}$$

Definition 3.2. A *Pure Type System with Ultra Σ-types* ($\text{PTS}_{\beta\sigma}$ or $\Sigma\text{-PTS}_{\beta\sigma}$) is defined as a Pure Type System with Σ-types extended with the rules shown in Figure 4.

For our second extension of Pure Type Systems, we replace the Σ-type by the same pair constructor used at the level of objects. We use the same notation of pairs for types and objects, i.e. we write $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,, \lambda x{:}A.P\rangle$. The σ-reduction and the typing rules are slightly diffferent. In Figure 5 we show the σ-reduction when Pairs are used. For the second extension the property which allows to do program extraction is stronger than for the first one: the reduction relation $\to_\sigma$ performs the task of splitting *any spec-pseudoterm* into a pair. This is because the assumption for spec-variables can be made stronger: we can associate a pair $\langle x_d\,, x_p\rangle$ to any spec-variable x_s. As above, in order to avoid the problems mentioned in [13], we restrict β-reduction to σ-normal forms.

Pair Type $(k_1, k_2, k_3) \in \mathcal{R}'$

$$\frac{\Gamma \vdash A{:}k_1 \quad \Gamma \vdash P{:}A \to k_2}{\Gamma \vdash \langle A\,, P\rangle{:}k_3}$$

Pair Object

$$\frac{\Gamma \vdash a{:}A \quad \Gamma \vdash p{:}P\,a \quad \Gamma \vdash \langle A\,, P\rangle{:}k}{\Gamma \vdash \langle a\,, p\rangle{:}\langle A\,, P\rangle}$$

Spec-variable

$$\frac{\Gamma \vdash x_d{:}A \quad \Gamma \vdash x_p{:}(P\,x_d) \quad \Gamma \vdash \langle A\,, P\rangle{:}k}{\Gamma \vdash x_s{:}\langle A\,, P\rangle}\ x_s \text{ is not in } \Gamma$$

σ-conversion

$$\frac{\Gamma \vdash u{:}U \quad \Gamma \vdash U'{:}k}{\Gamma \vdash u{:}U'}\quad U{=_\sigma}U'$$

Data-variable

$$\frac{\Gamma \vdash x_s{:}\langle A\,, P\rangle}{\Gamma \vdash x_d{:}A}\ x_d \text{ is not in } \Gamma$$

Prop-variable

$$\frac{\Gamma \vdash x_s{:}\langle A\,, P\rangle}{\Gamma \vdash x_p{:}P\,x_d}\ x_p \text{ is not in } \Gamma$$

Figure 6: Typing rules using only the pair constructor

Definition 3.3. Let $\mathsf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{R}')$ where $\mathcal{S}$ is a set of sorts, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of axioms, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of rules for the product and $\mathcal{R}' \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of rules for the pair constructor. The notion of *Pure Type System extended with Pairs* ($\mathrm{PTS}_{\beta\sigma}$ or P-$\mathrm{PTS}_{\beta\sigma}$) is inductively defined by adding the rules in Figure 6 to the rules of Pure Type Systems.

4 Systems with Ultra Σ-types and Pairs

In this section we present examples of Pure Type Systems with Ultra Σ-types and Pairs.

4.1 Ultra Σ in Martin-Löf's Type Theory

Martin Löf's Type Theory can be coded in a PTS_β called λP. Similarly, the Theory of Specifications of [14] – based on Martin Löf's Type Theory – can be coded in a $\mathrm{PTS}_{\beta\sigma}$ whose specification is, essentially three copies of λP:

Sorts	$\mathcal{S}$	$\star_\mathsf{d} \quad \star_\mathsf{p} \quad \star_\mathsf{s} \quad \square_\mathsf{d} \quad \square_\mathsf{p} \quad \square_\mathsf{s}$
Axioms	$\mathcal{A}$	$(\star_\mathsf{d}, \square_\mathsf{d}) \quad (\star_\mathsf{p}, \square_\mathsf{p}) \quad (\star_\mathsf{s}, \square_\mathsf{s})$
Rules for Π	$\mathcal{R}$	$(\star_\mathsf{u}, \star_\mathsf{v}) \quad (\star_\mathsf{u}, \square_\mathsf{u}) \quad \text{for } \mathsf{u}, \mathsf{v} \in \{\mathsf{d}, \mathsf{p}, \mathsf{s}\}$
		$(\star_\mathsf{d}, \square_\mathsf{p})$
Rules for Σ	$\mathcal{R}'$	$(\star_\mathsf{d}, \star_\mathsf{p}, \star_\mathsf{s})$

In order to obtain the full theory of [14] we also have to add several distinguished variables[2] with the following type declarations to the context:

$$\begin{array}{ll} \mathit{data} : \star_\mathsf{d} & \mathit{prog} : \mathit{data} \to \star_\mathsf{d} \\ \mathit{prop} : \star_\mathsf{p} & \mathit{proof} : \mathit{prop} \to \star_\mathsf{p} \\ \mathit{spec} : \star_\mathsf{s} & \mathit{impl} : \mathit{spec} \to \star_\mathsf{s} \end{array}$$

We also have to add a σ-reduction rule for *spec* and *impl*.

$$\begin{array}{lll} \mathit{spec} & \to_\sigma & \Sigma x{:}\mathit{data}.(\mathit{prog}\, x) \to \mathit{prop} \\ \mathit{impl}\ \langle a\,, p\rangle & \to_\sigma & \Sigma y{:}(\mathit{prog}\, a).\mathit{proof}(p\, y) \end{array}$$

On the other hand, the Theory of Specifications of [14] is more restrictive than this Pure Type System. The derivations $\Gamma \vdash u : U$ in this Pure Type System should be restricted as follows: [3]

[2] Note that these are constants, but in Pure Type Systems constants are treated as variables.

[3] These restrictions are similar to the ones that apply to the codification of Martin Löf's Type Theory in λP.

1. The only spec-variables $x_s : S : \Box_\mathsf{s}$ that occur in the context Γ are $spec : \star_\mathsf{s}$ and $impl : spec \to \star_\mathsf{s}$.
2. In the derivation of $\Gamma \vdash u : U$, there are no abstractions $\lambda x{:}U.v$ of type $\Pi x{:}U.V$ formed with the rules $(\star_\mathsf{d}, \Box_\mathsf{d})$, $(\star_\mathsf{p}, \Box_\mathsf{p})$ or $(\star_\mathsf{s}, \Box_\mathsf{s})$. Hence there are no β-redexes $((\lambda x{:}U.v)\, u)$ formed in this way.

Since the Theory of Specifications of [14] uses two different constructors, one pair constructor for types and another for objects [4], it is more appropiate to model this theory as a particular Pure Type System with Ultra Σ-types rather than a Pure Type System with Pairs. In [14] there is a presentation of a subset of the theory as a PTS but it lacks a notion of pairs.

The following result is the main property of the Theory of Specifications: the σ-normal form is a mapping from the Theory of Specification into a system called Verification Calculus (VC). VC is a PTS_β obtained from the specification defined above by removing the spec-sorts ($\star_\mathsf{s}$ and $\Box_\mathsf{s}$) and the product rule $(\star_\mathsf{p}, \star_\mathsf{d})$.

Theorem 4.1. (Program extraction). Assume $\Gamma \vdash u{:}U$ satisfies the two restrictions mentioned above.

1. If $\Gamma \vdash u{:}U{:}\star_\mathsf{d}$ or $\Gamma \vdash u{:}U{:}\star_\mathsf{p}$ then, $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} \mathsf{nf}_\sigma(u){:}\mathsf{nf}_\sigma(U)$.
2. If $\Gamma \vdash u{:}U{:}\star_\mathsf{s}$ then $\mathsf{nf}_\sigma(U) = \Sigma x{:}A.P$, $\mathsf{nf}_\sigma(u) = \langle a\,, p\rangle$, $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} a{:}A$ and $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} p{:}P[a/x]$.

This is proved by induction on the derivation.

4.2 Pairs in the Calculus of Constructions

The Theory of Specifications of [15, 5] is a $\mathrm{PTS}_{\beta\sigma}$ whose specification is defined as follows:

Sorts	$\mathcal{S}$	$\star_\mathsf{d}{}^\mathsf{i}\ \star_\mathsf{p}{}^\mathsf{i}\ \star_\mathsf{s}{}^\mathsf{i}$	for $\mathsf{i} \in N$
Axioms	$\mathcal{A}$	$(\star_\mathsf{d}{}^\mathsf{i}, \star_\mathsf{d}{}^{\mathsf{i}+1})\ (\star_\mathsf{p}{}^\mathsf{i}, \star_\mathsf{p}{}^{\mathsf{i}+1})\ (\star_\mathsf{s}{}^\mathsf{i}, \star_\mathsf{s}{}^{\mathsf{i}+1})$	for $\mathsf{i} \in N$
Rules for Π	$\mathcal{R}$	$(\star_\mathsf{u}^\mathsf{i}, \star_\mathsf{v}^\mathsf{j})$	for $\mathsf{i} \le \mathsf{j}$ or $\mathsf{j} = 0$ and $\mathsf{u}, \mathsf{v} \in \{\mathsf{d}, \mathsf{p}, \mathsf{s}\}$
Rules for Pair	$\mathcal{R}'$	$(\star_\mathsf{d}{}^\mathsf{i}, \star_\mathsf{p}{}^\mathsf{i}, \star_\mathsf{s}{}^\mathsf{i})$	for $\mathsf{i} \in N$

In order to obtain the full Theory of Specifications [15] we also have to add σ-rules for spec-sorts:

$$\star_\mathsf{s}{}^\mathsf{i} \to_\sigma \langle \star_\mathsf{d}{}^\mathsf{i}\,, \lambda x{:}\star_\mathsf{d}{}^\mathsf{i}.x \to \star_\mathsf{p}{}^\mathsf{i}\rangle$$

The Verification Calculus (VC) is a PTS_β obtained from the specification defined above by removing the spec-sorts $(\star_\mathsf{s}{}^\mathsf{i})$ and the product rules $(\star_\mathsf{p}{}^\mathsf{i}, \star_\mathsf{d}{}^\mathsf{j})$. The following theorem, which is proved by induction on the derivation, states the main property of this system (similar to the one in the previous section).

Theorem 4.2. (Program extraction). Let $\Gamma \vdash u{:}U$.

1. If u is a data or prop-pseudoterm then, $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} \mathsf{nf}_\sigma(u){:}\mathsf{nf}_\sigma(U)$.
2. If u is a spec-pseudoterm then $\mathsf{nf}_\sigma(U) = \langle A\,, P\rangle$, $\mathsf{nf}_\sigma(u) = \langle a\,, p\rangle$, $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} a{:}A$ and $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} p{:}P\, a$.

In [5] we show that the Theory of Specifications based on the Calculus of Constructions satisfies subject reduction and strong normalization when we restrict β-reduction to σ-normal forms. Counterexamples to subject reduction and strong normalization when β-reduction is not restricted are shown in [13].

[4] Hence distributivity of the abstraction and application is defined only on objects pairs.

4.3 Realizability interpretations for program extraction

In Coq until version 6.3 [1] the extraction procedure has been performed by means of an external function based on realizability interpretations [11, 10]. In [10] data types, propositions and specifications are distinguished with the sorts Data, Prop and Spec. They belong to the universes dType and Type. In the following specification we will shorten the names of the sorts. The system in [10] can be described as a $\mathrm{PTS}_{\beta\sigma}$ with the following specification:

$$
\begin{array}{ll}
\mathcal{S} & \star_{\mathsf{d}} \quad \star_{\mathsf{p}} \quad \star_{\mathsf{s}} \quad \Box_{\mathsf{d}} \quad \Box \\
\mathcal{A} & (\star_{\mathsf{d}}, \Box_{\mathsf{d}}) \quad (\star_{\mathsf{p}}, \Box) \quad (\star_{\mathsf{s}}, \Box) \\
\mathcal{R} & (k_1, k_2) \quad \text{for } k_1 \in \{\star_{\mathsf{d}}, \Box_{\mathsf{d}}\}, k_2 \in \{\star_{\mathsf{p}}, \star_{\mathsf{s}}, \Box\} \\
 & (k_1, k_2) \quad \text{for } k_1, k_2 \in \{\star_{\mathsf{p}}, \star_{\mathsf{s}}, \Box\} \\
 & (\star_{\mathsf{p}}, \star_{\mathsf{d}}) \\
\mathcal{R}' & (\star_{\mathsf{d}}, \star_{\mathsf{p}}, \star_{\mathsf{s}})
\end{array}
$$

Since the σ-reduction introduces some intermediate steps that were not present in the description of [10], we had to include the rule $(\star_{\mathsf{p}}, \star_{\mathsf{d}})$ in the above specification which was not present in the rules of [10]. Note that the rules for Σ-types are different from Coq. A Σ-type does not belong to $\star_{\mathsf{d}}$ but to $\star_{\mathsf{s}}$.

The distinction between data-pseudoterms, prop-pseudoterms and spec-pseudoterms is also made in [10]. Prop-pseudoterms are called non-informative terms and spec-pseudoterms are called informative terms. Though in this system $\Box_{\mathsf{p}}$ is identified with $\Box_{\mathsf{s}}$, the distinction made at syntactical level allows us to deduce the one at type level: if a kind is a prop-pseudoterm, then it should have type $\Box_{\mathsf{p}}$ and if the kind is a spec-pseudoterm, then it should have type $\Box_{\mathsf{s}}$. Hence, in both approaches the distinction between data types, propositions and specifications is necessary.

The system in [10] lacks the σ-conversion rule and the process of program extraction is described by two external functions $\mathcal{E}$ and $\mathcal{R}$ that compute the normal form of the σ-reduction. The σ-reduction extends $\mathcal{E}$ and $\mathcal{R}$ to type systems beyond $F\omega$ and CC [15]. On one hand the definition of σ-reduction is short and elegant. On the other hand it 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 $\langle \mathcal{E}(s), \mathcal{R}(s) \rangle$.

The extraction procedure in Coq version 7 [16] is also based on a reduction relation. There is one important difference: in all versions of Coq so far the extracted program is given in a programming language (ML) which is not part of the system whereas a $\mathrm{PTS}_{\beta\sigma}$ unifies the programming language and the logic.

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 $\mathcal{T}$. Using Ultra Σ-types the type extraction rule is derivable and expressed as: if $s : \Sigma x{:}A.P$ then $s \twoheadrightarrow_\sigma \langle a, p \rangle$ and the realizer a of P has type A (Theorem 4.1).

4.4 Specifications, Deliverables, Data Refinement

In LEGO [8, 17] a specification mechanism is implemented based on a pair of a type and a predicate over it [2, 9, 7]. The set of all specifications is described by an existential type in ECC [6], and a specification is a pair, but there are no objects of type pair. In this sense, this system can be thought of as a PTS with Σ types.

This notion of specification is used in [7] to formalize a method of program development by stepwise refinement in type theory, and it is used in [2, 9] to define a method of program extraction using the notion of deliverables. In [2, 9] a category is considered whose objects are specifications and morphisms are first order deliverables.

A subsystem of ECC is described as a PTS with Σ-types whose specification is given below:

$\mathcal{S}$	$\star^{\mathsf{i}}$	for $\mathsf{i} \in N$
$\mathcal{A}$	$(\star^{\mathsf{i}}, \star^{\mathsf{i}+1})$	for $\mathsf{i} \in N$
$\mathcal{R}$	$(\star^{\mathsf{i}}, \star^{\mathsf{j}}, \star^{\mathsf{n}})$	with $\mathsf{n} = max\{\mathsf{i},\mathsf{j}\}$ and $\mathsf{j} \neq 0$, or $\mathsf{j} = 0$ and $\mathsf{n} = 0$
$\mathcal{R}'$	$(\star^{\mathsf{i}}, \star^{\mathsf{j}}, \star^{\mathsf{n}})$	for $\mathsf{n} = max\{\mathsf{i},\mathsf{j}\}$.

In [6], the sort prop for propositions corresponds to $\star^0$ and the sorts Type^i corresponds to $\star^{i+1}$. We recall the notion of specification and deliverable given in [2]:

Definition 4.3. A *specification* is a 'pair' (A, P) consisting of a type $A : \star^{\mathsf{i}+1}$ and a function $P : A \to \star^0$.

Hence, a specification in [2] provides information for the formation of the Σ-type. In our approach, the notion of specification is more general, i.e. it is defined as any spec-pseudoterm of type $\star_{\mathsf{s}}$ or $\star_{\mathsf{s}}{}^{\mathsf{i}}$. For instance $\Pi x{:}\mathsf{Nat}.\langle A\,, P\rangle$ is a specification even though it starts with a Π constructor.

Definition 4.4. Let $S = (A, P)$, $T = (B, Q)$ and $U = (C, R)$ be specifications [5].

1. A *first order deliverable* (i.e. a morphism between the specifications S and T) is a term typable in ECC with the type $\mathsf{Del}_1\ S\ T = \Sigma f{:}(A \to B).\Pi x{:}A.(P\,x) \to Q\,(f\,x)$.
2. A *second order deliverable* is a term typable in ECC of type $\mathsf{Del}_2\ S\ T\ U = \Sigma f{:}A \to B \to C.\Pi x{:}A.((P\,x) \to \Pi y{:}B.(Q\,y) \to R\,(f\,x\,y))$.

Using Ultra Σ-types or Pairs, there is no need to distinguish between first and second order deliverables. Morphisms between specifications are written using the Π and λ of the inner lambda calculus. We use the type $\Pi x{:}S.T$ 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 in [2] 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:

Theorem 4.5. Let $S = \langle A\,, P\rangle$, $T = \langle B\,, Q\rangle$ and $U = \langle C\,, R\rangle$.

1. The σ-normal form of $(S \to T)$ is $\mathsf{Del}_1\ S\ T$, exactly the type of first order deliverables.
2. The type $\mathsf{Del}_2\ S\ T\ U$ of second order deliverables is the σ-normal form of an instance of the higher order type $\Pi x{:}S.(\Pi y{:}T.U)$ where S, T and U are dependent specifications.

In a system with Ultra Σ-types or Pairs we use the λ-abstraction to express functionality between specifications, the identity and composition are defined by the usual λ-terms whose σ-normal forms coincide with the identity and composition of first order deliverables.

Theorem 4.6.

1. The identity of first order deliverables is the σ-normal form of the identity in lambda calculus, i.e. $\lambda x{:}S.x$.
2. The composition of first order order deliverables is the σ-normal form of the composition in lambda calculus, i.e. $\lambda f{:}S \to T.\lambda g{:}T \to R.\lambda x{:}S.f(gx)$.

The refinement maps used in [7] to implement a specification can also be given as terms in our PTS with Ultra Σ or with Pairs. We can reformulate data refinement in a more uniform way using Pairs as follows. Let S, T be specifications, $S =_\sigma \langle A\,, P\rangle{:}\langle \star_{\mathsf{d}}\,, \lambda x{:}\star_{\mathsf{d}}.x \to \star_{\mathsf{p}}\rangle{:}\star_{\mathsf{s}}$, $T =_\sigma \langle B\,, Q\rangle{:}\langle \star_{\mathsf{d}}\,, \lambda x{:}\star_{\mathsf{d}}.x \to \star_{\mathsf{p}}\rangle{:}\star_{\mathsf{s}}$. We recall that a refinement map is a function $\rho{:}A \to B$ satisfying the condition $\Pi x{:}A.Px \to Q(\rho x)$. We can write the type $S \to T$ in a $\mathrm{PTS}_{\beta\sigma}$ with Pairs, and

[5] In our notation these specifications are written as $S = \langle A\,, P\rangle$, $T = \langle B\,, Q\rangle$ and $U = \langle C\,, R\rangle$.

compute its σ-normal form, which is a pair $\langle A \to B\,, \lambda\rho{:}A \to B.\Pi x{:}A.Px \to Q(\rho x)\rangle$. The notion of refinement map and implementation is therefore completely internalized in our system, and σ-reduction can be used to compute the satisfaction condition associated to a refinement map.

Here we repeat the comment on page 162 of [12]. Manipulating the Pairs inside ECC using Σ-types does not impose restrictions on the shape of programs and specifications. The system ECC does not separate the worlds of data-types, propositions and specifications. In particular programs may contain logical parts which are not erased and specifications could be used as data-types. The systems with Pairs have an essential property on specifications which makes program extraction always possible: *An inhabitant of a specification always reduces to a pair whose first component does not contain logical parts in it.* (See Theorem 4.1). The deliverables of [2] do not have this property. A first order deliverable is an inhabitant in ECC of a Σ-type. Hence a deliverable may or may not be a pair. In case it is a pair the first component may contain logical parts which the system ECC cannot erase.

4.5 Programming logic $\lambda\omega_L$

The programming logic $\lambda\omega_L$ of [12] (page 38) is defined as a Pure Type System with the following specification:

$$\begin{array}{ll}
\mathcal{S} & \star_{\mathsf{d}}\ \star_{\mathsf{p}}\ \Box_{\mathsf{d}}\ \Box_{\mathsf{d}} \\
\mathcal{A} & (\star_{\mathsf{d}}, \Box_{\mathsf{d}})\ (\star_{\mathsf{p}}, \Box_{\mathsf{p}}) \\
\mathcal{R} & (\star_{\mathsf{d}}, \star_{\mathsf{d}})\ \ (\star_{\mathsf{p}}, \star_{\mathsf{p}})\ \ (\star_{\mathsf{d}}, \star_{\mathsf{p}}) \\
 & (\Box_{\mathsf{d}}, \star_{\mathsf{d}})\ \ (\Box_{\mathsf{p}}, \star_{\mathsf{p}})\ \ (\Box_{\mathsf{d}}, \star_{\mathsf{p}})\quad (\Box_{\mathsf{d}}, \Box_{\mathsf{d}})\ \ (\Box_{\mathsf{p}}, \Box_{\mathsf{p}})\ \ (\Box_{\mathsf{d}}, \Box_{\mathsf{p}}) \\
 & (\star_{\mathsf{d}}, \Box_{\mathsf{p}})
\end{array}$$

In [12] the names $\star_{\mathsf{s}}$ and $\Box_{\mathsf{s}}$ are used instead of $\star_{\mathsf{d}}$ and $\Box_{\mathsf{d}}$.

In $\lambda\omega_L$ specifications are always Pairs (σ-normal forms) which get manipulated at the meta-level. The logic $\lambda\omega_L$ 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.

The notion of specification of [12] can be made explicit in the syntax using the pair constructor. A pair of rules of $\lambda\omega_L$ is coded as only one rule between specifications in our system. The reduction relation σ 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 our system gives a rule between Pairs that codes the pair of rules (the coupled derivation rules) in $\lambda\omega_L$.

The specification of the Pure Type System with Pairs that codes the system $\lambda\omega_L$ of [12] is given below.

$$\begin{array}{ll}
\mathcal{S} & \star_{\mathsf{d}}\quad \star_{\mathsf{p}}\quad \star_{\mathsf{s}}\quad \Box_{\mathsf{d}}\quad \Box_{\mathsf{d}}\quad \Box_{\mathsf{s}} \\
\mathcal{A} & (\star_{\mathsf{d}}, \Box_{\mathsf{d}})\ \ (\star_{\mathsf{p}}, \Box_{\mathsf{p}})\ \ (\star_{\mathsf{s}}, \Box_{\mathsf{s}}) \\
\mathcal{R} & (\star_{\mathsf{u}}, \star_{\mathsf{v}})\quad \text{for } \mathsf{u}, \mathsf{v} \in \{\mathsf{d}, \mathsf{p}, \mathsf{s}\} \\
 & (\Box_{\mathsf{d}}, \star_{\mathsf{d}})\ \ (\Box_{\mathsf{p}}, \star_{\mathsf{p}})\ \ (\Box_{\mathsf{d}}, \star_{\mathsf{p}})\ \ (\Box_{\mathsf{d}}, \star_{\mathsf{s}})\quad (\Box_{\mathsf{d}}, \Box_{\mathsf{d}})\ \ (\Box_{\mathsf{p}}, \Box_{\mathsf{p}})\ \ (\Box_{\mathsf{d}}, \Box_{\mathsf{p}}) \\
 & (\star_{\mathsf{d}}, \Box_{\mathsf{p}}) \\
\mathcal{R}' & (\star_{\mathsf{d}}, \star_{\mathsf{p}}, \star_{\mathsf{s}})
\end{array}$$

The product, abstraction and application rules instantiated with the set of rules $(\star_{\mathsf{s}}, \star_{\mathsf{s}})$ and $(\Box_{\mathsf{d}}, \star_{\mathsf{s}})$ code the coupled derivation rules on pages 45-48 of [12]. As an example we show the coupled derivation rules for the product formed with $(\star_{\mathsf{s}}, \star_{\mathsf{s}})$ and how to code them as only one rule in our system. The coupled derivation rules for functions formed with $(\star_{\mathsf{d}}, \star_{\mathsf{d}})$ given on page 46 of [12] are the pair of rules:

$$(1)\qquad \frac{\Gamma \vdash A, B{:}\star_{\mathsf{d}}}{\Gamma \vdash A \to B{:}\star_{\mathsf{d}}} \qquad\qquad \frac{\Gamma \vdash P{:}A \to \star_{\mathsf{p}} \quad \Gamma \vdash Q{:}A \to B \to \star_{\mathsf{p}}}{\Gamma \vdash R{:}(A \to B) \to \star_{\mathsf{p}}}$$

where R is the propositional function $\lambda f{:}A \to B.\Pi x_d{:}A.(P\,x_d) \to Q x_d\,(f\,x_d)$.

The conclusions of these two rules are put together into a pair of type $\star_{\mathsf{s}}$:

$$\Gamma \vdash \langle A \to B\,, \lambda f{:}A \to B.(R\,f)\rangle{:}\star_{\mathsf{s}}$$

This pair is the σ-normal form of the product $\Pi x_s{:}S.T$ formed with the rule $(\star_{\mathsf{s}}, \star_{\mathsf{s}})$ where $S = \langle A\,, P\rangle$ and $T = \langle B\,, \lambda x_d{:}B.Q\,x_d(f\,x_d)\rangle$. The type of that product is typable using the product rule instantiated with $(\star_{\mathsf{s}}, \star_{\mathsf{s}})$, i.e.

$$(2) \quad \frac{\Gamma \vdash S{:}\star_{\mathsf{s}} \quad \Gamma, x_s{:}S \vdash T{:}\star_{\mathsf{s}}}{\Gamma \vdash \Pi x_s{:}S.T{:}\star_{\mathsf{s}}}$$

Applying the σ-normal form to each judgement of (2) we obtain a derived rule between Pairs which codes the coupled derivation rules (1).

The σ-normal forms are a mapping from the $\mathrm{PTS}_{\beta\sigma}$ defined above into $\lambda\omega_L$. Hence "the Verification Calculus of Poll's system" is exactly $\lambda\omega_L$ (without coupled derivation rules). We have to add an extra rule of σ-reduction for spec-variables: $x_s{\to_\sigma}\langle x_d\,, x_p\rangle$. Moreover we add typing rules to be able to deduce that $x_s : \star_{\mathsf{s}}$ if and only if $x_d : \star_{\mathsf{d}}$ and $x_p : x_d \to \star_{\mathsf{p}}$.

Theorem 4.7. (Program extraction). Let $\Gamma \vdash u{:}U$.

1. If u is a data or prop-pseudoterm then, $\mathsf{nf}_\sigma(\Gamma) \vdash_{\lambda\omega_L} \mathsf{nf}_\sigma(u){:}\mathsf{nf}_\sigma(U)$.
2. If u is a spec-pseudoterm then $\mathsf{nf}_\sigma(U) = \langle A\,, P\rangle$, $\mathsf{nf}_\sigma(u) = \langle a\,, p\rangle$, $\mathsf{nf}_\sigma(\Gamma) \vdash_{\lambda\omega_L} a{:}A$ and $\mathsf{nf}_\sigma(\Gamma) \vdash_{\lambda\omega_L} p{:}P\,a$.

This is proved by induction on the derivation.

5 Axioms of Choice and Independence of Premisses

In this section we present another application of PTSs with Pairs: the derivation of logical axioms. Both the Axiom of Choice (AC) and Independence of Premises (IP) are derivable in this formalism. Again we abbreviate $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,, \lambda x{:}A.P\rangle : \star_{\mathsf{s}}$. AC is trivially deduced since it is given by a σ-reduction rule:

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

The axiom IP [4] is expressed as $S \to T$ where $S = (P \to \Sigma x{:}A.Q\,x)$ and $T = \Sigma x{:}A.(P \to Q\,x)$. We show that $S {\twoheadrightarrow_\sigma} T$. Using this we can trivially deduce that $S \to T$ is inhabited.

$$\begin{array}{rcl} S = (P \to \Sigma x{:}A.Q\,x) & \to_\sigma & \Sigma f{:}(P \to A).(\Pi x{:}P.Q\,(f\,x)) \\ & \twoheadrightarrow_\sigma & \Sigma f{:}A.(\Pi x{:}P.Q\,f) \\ & =_\alpha & \Sigma x{:}A.(P \to Q\,x) = T \end{array}$$

If S is typable then $S \to S$ is inhabited by the identity. Since $S =_\sigma T$, applying conversion rule, we conclude that the identity has also type $S \to T$. Hence the axiom IP is derivable using Ultra Σ-types. Note that the assumption in [4] that P is Harrop (or "self-realized" or "inhabited") is not needed because we have the possibility of erasing P.

The axiom of IP is not derivable in Martin Löf's Type Theory but it is (Kreisel modified) realizable for any Harrop formula P. This shows that the Theory of Specifications of [14] is not conservative over Martin Löf's Type Theory. This observation is not surprising. Conservativity holds over the Verification Calculus where data types and propositions are not identified.

As a last remark, note that $(\mathsf{False} \to \Sigma x{:}\mathsf{Empty}.Q\,x)$ is not derivable in a $\mathrm{PTS}_{\beta\sigma}$. From the proposition False, we can derive any proposition but we cannot derive a specification like $\Sigma x{:}\mathsf{Empty}.Q\,x$. This shows that specifications do not behave like propositions.

6 Conclusions

We have shown that σ-reduction is a way of describing the process of extracting a program which is independent of the typed lambda calculus we choose. An obvious conclusion from this paper is that any program that can be extracted from a proof of its specification using LEGO [2] or Coq [10] can also be extracted using σ-reduction. It will be interesting to find an example of program extraction that can be handled by σ-reduction but cannot be done in LEGO or Coq.

References

[1] Barras et al. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 1999.

[2] 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*, 1990.

[3] James L. Caldwell. Classical Propositional Decidability via Nuprl Proof Extraction. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOL'98), Australia*, pages 105–122. Springer, 1998.

[4] T. Crolard. A type theory which is complete for Kreisel's modified realizability. *ENTCS*, 23(1), 1999.

[5] M. Fernández and P. Severi. An operational approach to program extraction in the Calculus of Constructions. In *Proc. of the Int. Workshop on Logic Based Program Development and Transformation (LOPSTR'02)*. Springer, 2002. LNCS.

[6] Z. Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of LICS'89*, IEEE, pages 386–395. IEEE Computer Society Press, 1989.

[7] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3:333–363, 1993.

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

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

[10] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

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

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

[13] F. van Raamsdonk and P. Severi. Eliminating proofs from programs. In *Proc. 3rd. International Workshop on Logical Frameworks and Meta-Languages*, volume 70.2 of *ENTCS*, 2002.

[14] 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.

[15] P. Severi and N. Szasz. Internal Program Extraction in the Inductive Calculus of Constructions. In *Proceedings of WAIT'02, 31st JAIIO*, 2002.

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

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