Internal Program Extraction in the Calculus of Inductive Constructions

Paula Severi[1] and Nora Szasz[2] *

[1] Departimento di Informatica, Università di Torino, Torino, Italy
[2] Instituto de Computación, Facultad de Ingeniería, Montevideo, Uruguay

Abstract. Based on the Calculus of Constructions extended with inductive definitions we present a Theory of Specifications with rules for simultaneously constructing programs and their correctness proofs. The theory contains types for representing specifications, whose corresponding notion of implementation is that of a pair formed by a program and a correctness proof. The rules of the theory are such that in implementations the program parts appear mixed together with the proof parts. A reduction relation performs the task of separating programs from proofs. Consequently, every implementation computes to a pair composed of a program and a proof of its correctness, and so the program extraction procedure is immediate.

1 Introduction

In Coq [Bar99], the set construction $\{\mathtt{x} : \mathtt{A} | (\mathtt{P\ x})\}$ is used to write specifications formed by a set A and a propositional function P over A. Then from an element in a set of the form $(x : A)\{\mathtt{y} : \mathtt{B} | (\mathtt{P\ x\ y})\}$ a program of type $A \rightarrow B$ can be extracted by means of an external function based on realizability interpretation [PM89]. The extracted program obtained in Coq does not belong to the system itself and hence it cannot be manipulated inside Coq.
In [Sza97] and [SS01] a Theory of Specifications with built-in program extraction is presented based on Martin Löf's Type Theory. This calculus internalizes the notion of realizability. Besides data types (sets) and propositions, we have a third sort of types for specifications. The essential property of a specification is that it *computes* to a pair formed by a data type and a predicate over it. This concept of pair is not defined as a mere existential or inductive type since it has its own particular reduction rules. Applying these rules, every object of a specification computes to a pair composed by a program and its correctness proof. The process of program extraction consists in first computing the normal form of this reduction and then taking the first projection. We can also extract the correctness proof internally by applying the second projection.
The aim of this paper is to extend the Theory of Specifications to include inductive types like natural numbers in order to be able to write useful examples and extract real programs. The idea in [SS01] of defining a reduction relation that captures the process of program extraction will be adapted to the framework

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

of the Calculus of Constructions and extended with inductive data types and inductive propositions.
Next we describe briefly the structure of this paper. In section 2 we present the Verification Calculus as two copies of the same type system, one for programs and one for proofs. In section 3 we present the Theory of Specifications: we introduce the syntax, the reduction relation $\rightarrow_\sigma$ and the type system of the theory. In section 4 we show an example of how to extract a program using σ-reduction. In section 5 we extend the Theory of Specifications with inductive data types and propositions. In section 6 we prove that the σ-reduction is confluent and normalizing. We then show how the projections can be encoded in this theory and we prove that the function nf_σ that computes the σ-normal form is a mapping from the Theory of Specifications into the Verification Calculus. Finally, in section 7 we suggest some directions for future research.

2 Verification Calculus

In this section we define the Verification Calculus. This calculus contains two copies of the Calculus of Constructions extended with an infinite type hierarchy [Luo89,Bar99], one for data types and one for propositions.
We assume the reader to be familiar with Pure Type Systems [Bar92] and just recall the typing rules.

Definition 2.1. Let $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a specification, i.e. a set $\mathcal{S}$ of sorts, a set $\mathcal{A}$ of axioms and a set $\mathcal{R}$ of rules. A Pure Type System (PTS) is defined by the rules shown in Figure 1.

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

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

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

Product
$$\frac{\Gamma \vdash U{:}k_1 \quad \Gamma, x{:}U \vdash V{:}k_2}{\Gamma \vdash \Pi x{:}U.V{:}k_3}\ (k_1, k_2, k_3) \in \mathcal{R}$$

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

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

Fig. 1. Typing rules for Pure Type Systems

The specification ECC for the Extended Calculus of Constructions. The extended Calculus of Constructions [Luo89] is obtained as a PTS with the following specification ECC.

1. Sorts. $Type^n$ for all $n \in \mathbb{N}$.
2. Axioms. $Type^n : Type^{n+1}$ for all $n \in \mathbb{N}$.
3. Rules. $(Type^n, Type^0, Type^0)$ $(Type^n, Type^m, Type^m)$ for $n \leq m$.

Duplicating ECC for the Verification Calculus. The Verification Calculus, used to prove the soundness of the Theory of Specifications, can be defined as a PTS with specification VC consisting of two copies of ECC, one for data types and the other for propositions, in the following way:

1. Sorts. The sorts are either data-sorts $dType^i$ or prop-sorts $pType^i$ for $i \in \mathbb{N}$. Note that the sorts in VC are obtained by concatenating the letter d or p to the sorts of ECC. We write dk for the concatenation of the letter d with the sort k and pk for the concatenation of the letter p with the sort k.
2. Axioms. For each axiom $k : k'$ in ECC we add two axioms in the Verification Calculus:

$$dk : dk' \qquad pk : pk'.$$

3. Rules. For each rule (k_1, k_2, k_3) in ECC, we add three rules in the Verification Calculus:

$$(dk_1, dk_2, dk_3) \qquad (dk_1, pk_2, pk_3) \qquad (pk_1, pk_2, pk_3)$$

Since in the Verification Calculus there are no rules of the form (pk_1, dk_2, dk_3) data types cannot depend on propositions.

3 Theory of Specifications

In this section we define the Theory of Specifications as a PTS (with the specification TS defined below) extended with pairs.

Definition 3.1. The specification TS (sorts, axioms and product rules) for the Theory of Specifications is defined as follows:

1. Sorts. The sorts in the Theory of Specifications are either data-sorts $dType^i$, prop-sorts $pType^i$ or spec-sorts $sType^i$ for $i \in \mathbb{N}$. Note that for each sort k in ECC, the data-sorts are obtained as the concatenation of d with k (written dk), similarly we obtain the prop-sorts and the spec-sorts as pk and sk.
2. Axioms. For each axiom $k : k'$ in ECC we add three axioms in the Theory of Specifications:

$$dk : dk' \qquad pk : pk' \qquad sk : sk'$$

3. Rules. For each rule (k_1, k_2, k_3) in ECC, we add the following rules:

$$(uk_1, vk_2, vk_3)$$

where u and v are either d, p or s.

We define the sort for data types: $data{:=}dType^0$, the sort for propositions: $prop{:=}pType^0$ and the sort for specifications: $spec{:=}sType^0$.

Note that in the specification TS defined above, we have rules of the form (pk_1, dk_2, dk_3) that allow the formation of data types dependending on propositions and hence, programs may contain proofs in it.
In order to be able to give a definition of the reduction $\rightarrow_\sigma$ which does not depend on the typing relation, we distinguish between data-pseudoterms, prop-pseudoterms and spec-pseudoterms (Figure 2).
We split the set of variables into three: data-variables (denoted by $x_d, y_d \ldots$), prop-variables (denoted by $x_p, y_p \ldots$) and spec-variables (denoted by $x_s, y_s \ldots$). An arbitrary pseudoterm is denoted by $u, v, U, V \ldots$ as in the previous section.

Data-pseudoterms
$A := x_d \mid dk$
$\quad \mid \lambda x_d{:}A.A \mid \lambda x_p{:}P.A \mid \lambda x_s{:}S.A$
$\quad \mid A\,A \mid A\,P \mid A\,S$
$\quad \mid \Pi x_d{:}A.A \mid \Pi x_p{:}P.A \mid \Pi x_s{:}S.A$

Prop-pseudoterms
$P := x_p \mid pk$
$\quad \mid \lambda x_d{:}A.P \mid \lambda x_p{:}P.P \mid \lambda x_s{:}S.P$
$\quad \mid P\,A \mid P\,P \mid P\,S$
$\quad \mid \Pi x_d{:}A.P \mid \Pi x_p{:}P.P \mid \Pi x_s{:}S.P$

Spec-pseudoterms
$S := x_s \mid sk \mid \langle A\,, P \rangle$
$\quad \mid \lambda x_d{:}A.S \mid \lambda x_p{:}P.S \mid \lambda x_s{:}S.S$
$\quad \mid S\,A \mid S\,P \mid S\,S$
$\quad \mid \Pi x_d{:}A.S \mid \Pi x_p{:}P.S \mid \Pi x_s{:}S.S$

Contexts
$\Gamma := ()$
$\quad \mid \Gamma, x_d{:}A$
$\quad \mid \Gamma, x_p{:}P$
$\quad \mid \Gamma, x_s{:}S$

Fig. 2. Syntax for the Theory of Specifications

We define the reductions $\rightarrow_\sigma$ and $\rightarrow_\beta$ on pseudoterms. The first reduction gives an operational semantics to program extraction. The essential property of the notion of specification is that it always computes to a pair. The reduction relation $\rightarrow_\sigma$ performs the task of splitting spec-pseudoterms into pairs. For the latter to hold, a spec-variable should also reduce to a pair. For this, from now on we assume that there exists a function Ψ such that $\Psi(x_s) = \langle x_d\,, x_p \rangle$ for all spec-variables x_s. The function Ψ should be injective in each component.

Definition 3.2. We define $\rightarrow_\sigma$ as the least relation on pseudoterms that contains the rules shown in Figure 3.

We denote $\twoheadrightarrow_\sigma$ the reflexive, symmetric and transitive closure of $\rightarrow_\sigma$.

Splitting

$$x_s \to_\sigma \langle x_d\,, x_p\rangle$$
$$sType^n \to_\sigma \langle dType^n\,, \lambda x_d{:}dType^n.(x_d \to pType^n)\rangle$$

Eliminating proofs from programs

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

Currifying

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

Distributivity

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

Fig. 3. Definition of σ-reduction

Definition 3.3. We define the reduction $\to_\beta$ as the least relation on pseudoterms that contains the following rules:

$(\lambda x_d{:}A.u)\,a \to_\beta u[a/x_d]$
$(\lambda x_p{:}P.u)\,p \to_\beta u[p/x_p]$
$(\lambda x_s{:}S.u)\,s \to_\beta u[s/x_s]$

Definition 3.4. The Theory of Specifications is inductively defined by adding the rules shown in Figure 4 to the ones in Figure 1 for Pure Type Systems with the fixed specification TS (Definition 3.1).

Remark 3.5. The following notion of *completeness* is needed to restrict substitution (this is used in the β-rule and in the Application rule). We also need a modified definition of *freshness* with respect to a typing context.

- 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 is performed $u[v/x]$ we require that the variable x is complete for u.
In the following example we show that we may loose confluence if we do not assume that the variable is complete.

$$\begin{array}{ccc}
(\lambda x_s{:}\langle A\,, P\rangle.x_s)\,\langle a\,, p\rangle & \twoheadrightarrow_\sigma & (\lambda x_d{:}A.\lambda x_p{:}(P\,x_d).x_s)\,a\,p \\
\downarrow_\beta & & \downarrow_\beta \\
\langle a\,, p\rangle & & (\lambda x_p{:}(P\,a).x_s)\,p \\
|| & & \downarrow_\beta \\
\langle a\,, p\rangle & & x_s
\end{array}$$

These last two terms cannot be joined.

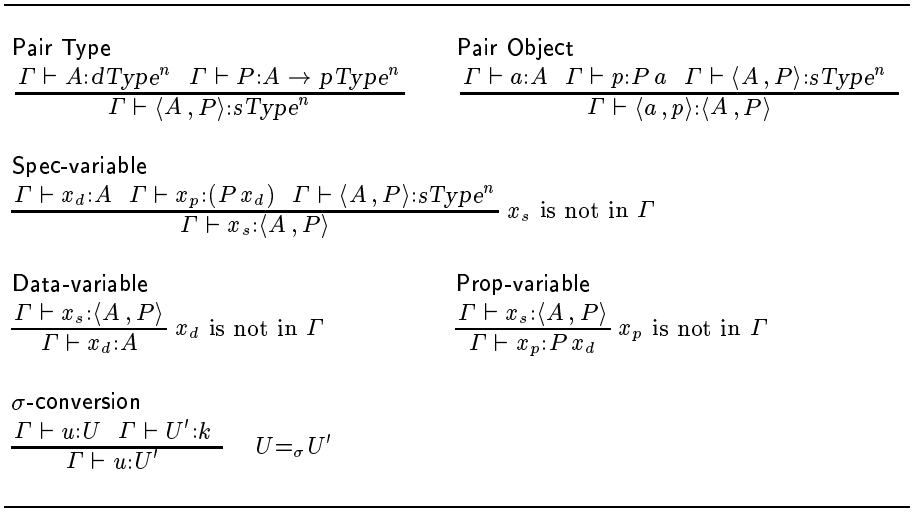

Fig. 4. Typing rules for pairs

4 An example of program extraction

In order to develop an example of program extraction, we illustrate how σ-reduction is extended to deal with natural numbers. Figure 5 shows the typing and reduction rules for natural numbers. There are in total 6 elimination rules. When $i = 0$ (k is either $dType^0, pType^0$ or $sType^0$) we can define functions by induction on natural numbers where the type of the result is a data type, a proposition or a specification. When $i = 1$ (k is either $dType^1, pType^1$ or $sType^1$) we have three elimination rules (called strong elimination rules) that allow us to define data types, propositions and specifications by induction on natural numbers. Strong elimination is necessary to prove, for instance, that $0 \neq (\mathsf{succ}\,0)$.
The σ-reduction is extended with a distribituvity rule. Note that the predicate $\hat{P}$ is in fact the predicate P applied to the first component of the pair.
In the next section we generalize this extension for a wide class of inductive data types and inductive propositions.
We are now ready to give an example of program extraction in the Theory of Specifications. We consider the specification stating that every natural number not equal to zero has a predecessor. Using $\Sigma x{:}A.P \ =^{\mathsf{def}}\ \langle A\,,P\rangle : spec$, this specification is written as follows:

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

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

- $A = \lambda n{:}\mathsf{Nat}.\mathsf{Nat}$,

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

Then, the following term is an inhabitant of the type S (for the sake of exposition we omit the first parameter of natrec):

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

Using σ-reduction, s is reduced to a pair. The first component is the extracted program, that is, the predecesor function, and the second component is the proof of its correctness.

$$s \twoheadrightarrow_\sigma \langle \underbrace{\lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ 0\, (\lambda m{:}\mathsf{Nat}.\lambda x_d{:}\mathsf{Nat}.m)\, n}_{\text{predecesor}}, \lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ p_0\, q\, n \rangle$$

We assume $\mathsf{Nat} : \mathit{data}$ and $\mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}$.

Elimination

$$\frac{\Gamma \vdash n{:}\mathsf{Nat} \quad \Gamma \vdash U{:}\mathsf{Nat} \to k \quad \Gamma \vdash u_1{:}(U\, 0) \quad \Gamma \vdash u_2{:}\Pi m{:}\mathsf{Nat}.((U\, m) \to (U\, (\mathsf{succ}\, m)))}{\Gamma \vdash (\mathsf{natrec}\ U\ u_1\ u_2\ n){:}(U\, n)}$$

where k is either $dType^i, pType^i$ or $sType^i$ for $i = 0, 1$.

Primitive recursion

$$(\mathsf{natrec}\ U\ u_1\ u_2\ 0) \to_\iota u_1$$
$$(\mathsf{natrec}\ U\ u_1\ u_2\ (\mathsf{succ}\, m)) \to_\iota (u_2\, m\, (\mathsf{natrec}\ U\ u_1\ u_2\ m))$$

Distributivity

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

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

---

Fig. 5. Natural Numbers

5 Inductive Types

In this section we extend the Verification Calculus and the Theory of Specifications with Inductive Types. These extensions are based on the Calculus of Inductive Constructions introduced in [Wer94].

Following [Wer94], inductive types (data types or propositions) are represented by terms of the form $\mathsf{Ind}(X{:}k)\{C_1(X)\dots C_n(X)\}$. We use the abbreviation $I = \mathsf{Ind}(X{:}k)\{C_1(X)\dots C_n(X)\}$. To simplify our presentation, we do not consider inductive types with parameters, i.e. we assume that $k{=}data$ or $k{=}prop$. The pseudoterms $C_1(X)\dots C_n(X)$ are the types of the constructors. For example, $\mathsf{Nat}{=}\mathsf{Ind}(X{:}data)\{X, X \to X\}$.
We denote the constructor in the position i of the inductive type I by $\mathsf{constr}(i, I)$ with $1 \leq i \leq n$. For instance $0{=}\mathsf{constr}(1, \mathsf{Nat})$ and $\mathsf{succ}{=}\mathsf{constr}(2, \mathsf{Nat})$. The elimination constant for the inductive type I (data type or proposition) is written as $\mathsf{elim}(I, U, v)\{u_1 \dots u_n\}$. The pseudoterm v is the object to be eliminated and it has type I. The pseudoterms $u_1, \dots, u_n$ are the branches which are applied depending on the constructor. For instance $(\mathsf{natrec}\ U\ u_1\ u_2\ n) = \mathsf{elim}(\mathsf{Nat}, U, n)\{u_1 u_2\}$.
In Figure 6 we give the typing rules for inductive data types ($k = data$) and inductive propositions ($k = prop$). The parameter $\mathcal{D}(k)$ is similar to the set $\mathcal{R}$ in the rules for pure type systems and it is used to forbid the elimination of a data type from a proposition: we impose the restriction that the set $\mathcal{D}(prop)$ contains neither data-sorts nor spec-sorts. In Coq there is a similar restriction.
The functions Δ and ρ used in the elimination, primitive recursion and distributivity rules will be defined later (Definition 5.5, Definition 5.6 and Definition 5.7).

Definition 5.1. The Verification Calculus with Inductive Types is obtained from the Verification Calculus by adding the rules for inductive types (Figure 6) instantiated with the sets $\mathcal{D}(data)$ and $\mathcal{D}(prop)$ given in the following table:

k	$\mathcal{D}(k)$
$data$	$dType^i \quad pType^i \quad i = 0, 1$
$prop$	$pType^i \quad i = 0, 1$

Definition 5.2. The Theory of Specifications with Inductive Types is obtained from the Theory of Specifications by adding the rules for inductive types (Figure 6) instantiated with the sets $\mathcal{D}(data)$ and $\mathcal{D}(prop)$ given in the following table:

k	$\mathcal{D}(k)$
$data$	$dType^i \quad pType^i \quad sType^i \quad i = 0, 1$
$prop$	$pType^i \quad i = 0, 1$

The type of the constructors should satisfy the restrictions given below to avoid inconsistences. Moreover, in the strong elimination rules the constructors should be small [Wer94].

Definition 5.3. We say that X is strictly positive in $\Pi \boldsymbol{x}{:}\boldsymbol{V}.X$ [1] if X does not belong to the free variables of $\boldsymbol{V}$ and it is not any of the bound variables $\boldsymbol{x}$.

[1] We abbreviate $v_1 \dots v_m$ by v and $\Pi x_1{:}V_1 \dots x_m{:}V_m.U$ by $\Pi \mathrm{x}{:}\mathrm{V}.U$.

We assume $I = \mathsf{Ind}(X{:}k)\{C_1(X) \ldots C_n(X)\}$ where k is either *data* or *prop*.

Formation

$$\frac{\Gamma, X{:}k \vdash C_i(X){:}k \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash I{:}k}$$

Introduction

$$\frac{\Gamma \vdash I{:}k}{\Gamma \vdash \mathsf{constr}(i, I){:}C_i(I)} \; 1 \leq i \leq n$$

Elimination

$$\frac{\begin{array}{l}\Gamma \vdash I{:}k \quad \Gamma \vdash U{:}I \to k', \quad k' \in \mathcal{D}(k) \\ \Gamma \vdash v{:}I \quad \Gamma \vdash u_j{:}\Delta\{C_j(I), U, \mathsf{constr}(j, I)\} \; \forall j, 1 \leq j \leq n\end{array}}{\Gamma \vdash \mathsf{elim}(I, U, v)\{u_1 \ldots u_n\}{:}(U\ v)}$$

Primitive Recursion

$$\mathsf{elim}(I, U, \mathsf{constr}(i, I)\mathrm{w})\{u_1 \ldots u_n\} \to_\iota (\Delta[C_i(I), u_i, F]\,\mathrm{w})$$

where $F = \lambda x{:}I.\mathsf{elim}(I, U, x)\{u_1 \ldots u_n\}$,
$\mathrm{w} = w_1 \ldots w_m$.

Distributivity

$$\mathsf{elim}(I, \langle A\,, P\rangle, v)\{\langle a_1\,, p_1\rangle \ldots \langle a_n\,, p_n\rangle\} \to_\sigma \langle \mathsf{elim}(I, A, v)\{a_1 \ldots a_n\}\,, \mathsf{elim}(I, \hat{P}, v)\{\hat{p_1} \ldots \hat{p_n}\}\rangle$$

where $\hat{P} = \lambda x{:}I.(P\,x\,\mathsf{elim}(I, A, x)\{a_1 \ldots a_n\})$,
$\hat{p_i} = \rho[C_i(I), p_i, \lambda x{:}I.\mathsf{elim}(I, A, x)\{a_1 \ldots a_n\}]$

Fig. 6. Typing rules for Inductive Types

Definition 5.4. A type constructor $C(X)$ is inductively defined by the following clauses:

1. $C(X) = X$,
2. $C(X) = \Pi x{:}U.C(X)$ where $C(X)$ is a type constructor, X does not belong to the free variables of U and $X \neq x$,
3. $C(X) = U \to C(X)$ where $C(X)$ is a type constructor and X is strictly positive in U.

The function $\Delta\{\ \}$ used in the elimination rule (Figure 6) gives the type of a branch u_j and it is defined as follows.

Definition 5.5. We define $\Delta\{C(X), U, c\}$ by induction on $C(X)$.
$\Delta\{X, U, c\} = (U\,c)$
$\Delta\{\Pi x{:}V.C(X), U, c\} = \Pi x{:}V.\Delta\{C(X), U, (c\,x)\}$
$\Delta\{(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X) \to C(X), U, c\} =$
$\quad \Pi y{:}(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X).(\Pi \boldsymbol{x}{:}\boldsymbol{V}.U(y\ \boldsymbol{x})) \to \Delta\{C(X), U, (c\,y)\}$

For instance, for Nat we have:

$$\Delta\{X, U, 0\} = (U\,0)$$
$$\Delta\{X \to X, U, \mathsf{succ}\} = \Pi n{:}X.(U\,n) \to (U\,(\mathsf{succ}\,n))$$

The function $\Delta[\ ]$ used in the primitive recursion scheme (Figure 6) is defined as follows.

Definition 5.6. We define $\Delta[C(X), f, F]$ by induction on $C(X)$.
$\Delta[X, f, F] = f$
$\Delta[\Pi x{:}V.C(X), f, F] = \lambda x{:}V.\Delta[C(X), (f\,x), F]$
$\Delta[(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X) \to C(X), f, F] =$
$\quad \lambda y{:}(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X).\Delta[C(X), (f\ y\ (\lambda \boldsymbol{x}{:}\boldsymbol{V}.F(y\boldsymbol{x}))), F]$

For instance, for Nat we have:

$$\Delta[X, f, F] = f,$$
$$\Delta[X \to X, f, F] = \lambda x{:}X.(f\ x\ (F x)).$$

The function ρ used in the distributivity rule (Figure 6) is defined as follows.

Definition 5.7. We define $\rho[C(X), p, F]$ by induction on $C(X)$.
$\rho[X, p, F] = p$
$\rho[\Pi x{:}V.C(X), p, F] = \lambda x{:}V.\rho[C(X), p\,x, F]$
$\rho[(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X) \to C(X), p, F] =$
$\quad \lambda y{:}(\Pi \boldsymbol{x}{:}\boldsymbol{V}.X).\lambda z{:}(\Pi \boldsymbol{x}{:}\boldsymbol{V}.\hat{P}(y\ \boldsymbol{x})).\rho[C(X), p\ y\ (\lambda \boldsymbol{x}{:}\boldsymbol{V}.F(y\boldsymbol{x}))z, F]$

For instance, for Nat we have:

$$\rho[X, p, F] = p,$$
$$\rho[X \to X, p, F] = \lambda x{:}X.\lambda y{:}\hat{P}x.(p\ x\ (F x)\ y).$$

6 Properties

In this section we prove some properties of the Theory of Specifications with Inductive Types.

6.1 Soundness

Lemma 6.1. *$\rightarrow_\sigma$ is confluent and normalizing.*
To prove that it is confluent we show that the parallel σ-reduction satisfies the diamond property. Normalization is proved by induction on the structure of the term.

By the previous lemma, the σ-normal form of a pseudoterm u exists and it is unique. We denote it as $\mathsf{nf}_\sigma(u)$.

Lemma 6.2. Let s be a spec-pseudoterm. Then, $\mathsf{nf}_\sigma(s) = \langle a\,, p\rangle$.

Substitution lemma holds excluding the problematic cases mentioned in remark 3.5.

Lemma 6.3.

1. $\mathsf{nf}_\sigma(u[a/x_d]) = \mathsf{nf}_\sigma(u)[\mathsf{nf}_\sigma(a)/x_d]$.
2. $\mathsf{nf}_\sigma(u[p/x_p]) = \mathsf{nf}_\sigma(u)[\mathsf{nf}_\sigma(p)/x_p]$.
3. $\mathsf{nf}_\sigma(u[s/x_s]) = \mathsf{nf}_\sigma(u)[a/x_d][p/x_p]$ where $\mathsf{nf}_\sigma(s) = \langle a\,, p\rangle$.

In the next lemma we show that the σ-normal form preserves β-reduction steps.

Lemma 6.4. If $u \rightarrow_{\beta\iota} u'$ then $\mathsf{nf}_\sigma(u) \twoheadrightarrow_{\beta\iota} \mathsf{nf}_\sigma(u')$.
The proof is done by induction on the structure of the term.

As an immediate consequence of the previous lemmas and theorems we obtain the following theorem.

Theorem 6.5. *$\rightarrow_{\beta\iota\sigma}$ is confluent.*

We prove that the Verification Calculus (with Inductive Types) is a syntactic model for the Theory of Specifications (with Inductive Types). The interpretation function is given by the mapping nf_σ. Derivation in the Theory of Specifications (with Inductive Types) is denoted by $\vdash$ and in the Verification Calculus (with Inductive Types) is denoted by $\vdash_{VC}$.

Theorem 6.6. (Soundness). 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 $\mathsf{nf}_\sigma(\Gamma) \vdash_{VC} p{:}P\,a$

The proof is by induction on the derivation.

6.2 Projections

In this section we show how to encode the projections in the Theory of Specifications and prove that a variant of the application rule is derivable.
Figure 7 shows the encoding of the projections in the Theory of Specifications.

Projections for spec-sorts	Projections for arbitrary spec-pseudoterms
$\pi_D = \lambda x_s{:}sType^i.x_d$	$\pi_d = \lambda x_s{:}sType^i.\lambda y_s{:}x_s.y_d$
$\pi_P = \lambda x_s{:}sType^i.x_p$	$\pi_p = \lambda x_s{:}sType^i.\lambda y_s{:}x_s.y_p$

Fig. 7. Codification of the projections for $i \in I\!N$.

Lemma 6.7.

1. $\pi_D \langle A\,, P\rangle \twoheadrightarrow_{\beta\sigma} A$ and $\pi_P \langle A\,, P\rangle \twoheadrightarrow_{\beta\sigma} P$.
2. $\pi_d \langle A\,, P\rangle \langle a\,, p\rangle \twoheadrightarrow_{\beta\sigma} a$ and $\pi_p \langle A\,, P\rangle \langle a\,, p\rangle \twoheadrightarrow_{\beta\sigma} p$.

Lemma 6.8.

1. If $\Gamma \vdash S{:}sType^i$ then $\Gamma \vdash \pi_D\, S{:}dType^i$ and $\Gamma \vdash \pi_P\, S{:}(\pi_D\, S) \to pType^i$.
2. If $\Gamma \vdash s{:}S$ then $\Gamma \vdash \pi_d\, S\, s{:}\pi_D\, S$ and $\Gamma \vdash \pi_p\, S\, s{:}\pi_P\, S(\pi_d\, S\, s)$.

Proof. It is easy to derive that π_D has type $sType^i \to dType^i$ and that π_P has type $\Pi x_s{:}sType^i.(x_d \to pType^i)$. Note that $\pi_D\, x_s =_{\beta\sigma} x_d$. Then, by applying conversion rule, we obtain that π_P has also type $\Pi x_s{:}sType^i.((\pi_D x_s) \to pType^i)$

Lemma 6.9.

1. $S =_{\beta\sigma} \langle \pi_D\, S\,, \pi_P\, S\rangle$.
2. $s =_{\beta\sigma} \langle \pi_d\ S\ s\,, \pi_p\ S\ s\rangle$

This is proved using Lemma 6.2.
When x is not complete for u, the substitution $u[v/x]$ is now allowed (see Remark 3.5). We will prove that there exists $u' =_{\beta\sigma} u$ such that x is complete for u' and, then, the substitution in u' can be performed.

Definition 6.10. For each spec-variable x_s, we define the reduction $\to_{xs}$ as the least relation on pseudoterms that is closed under the rule $x_s \to_{xs} \langle x_d\,, x_p\rangle$. We denote the normal form of this reduction as nf_{x_s}.

Definition 6.11. We define u^x by cases:

$$u^x = \begin{cases} u[(\pi_d\, S\, x_s)/x_d][(\pi_p\, S\, x_s)/x_p] & \text{if } x = x_s \\ \mathsf{nf}_{x_s}(u) & \text{if } x = x_d \text{ or } x = x_p \end{cases}$$

The variable x is complete for u^x and the substitution $u^x\ [v/x]$ is allowed (see Remark 3.5).

Lemma 6.12.

1. If $\Gamma, x{:}V, \Delta \vdash u{:}U$ then $\Gamma, x{:}V,\ \Delta^x\ \vdash\ u^x : U^x$.
2. $u^x\ =_{\beta\sigma} u$

The first part is proved by induction on the derivation and the second one by induction on the structure of u.

Theorem 6.13. The following variant of the application rule is derivable.

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

Proof: By the previous lemma we have that $\Gamma \vdash v{:}\Pi x{:}U.\ V^x$

7 Conclusions

We mention some differences between the Theory of Specifications and Coq. First in Coq there is only one type hierarchy of universes $Type^i$ instead of three $dType^i, pType^i$ and $sType^i$. Also in Coq there is a cumulativity rule, if $U{:}prop$ then $U{:}Type^{i+1}$ that we do not have in the Theory of Specifications. As a consequence, the status of an object of type $Type^{i+1}$ with respect to extraction is not clear in Coq. The Theory of Specifications, however, has interesting properties concerning extraction (Theorem 6.6). Due to these differences, it is not possible to implement the Theory of Specifications within Coq. A tool based on the Theory of Specifications should be implemented independently of Coq.
The Theory of Specifications is related to other programming logics that allow the derivation of implementations as pairs program-proof developed in parallel. Besides Coq, we also mention the Deliverables [BM90] and the programming logic $\lambda\omega_L$ [Pol94]. In the theory of Deliverables [BM90], a specification is a pair consisting of a data type and a predicate over it, and the definition of deliverable corresponds in a certain way to our notion of functions between specifications. Σ-types are used to put together both the components of specifications and the functions between them. This makes it problematic to obtain a good definition of function in a direct way. To overcome this difficulty second order deliverables have to be defined using a global specification as a parameter. In fact, the need of having second order deliverables arises from not having defined what a specification depending on another specification is. We think that to avoid these problems, it is essential to have a special construct for specifications, different from the Σ-types and to give special meaning to the dependences between specifications. Poll's $\lambda\omega_L$[Pol94] is a subsystem of the Verification Calculus (see Section 2). However, in [Pol94] the notions of specification and implementation are not completely formalized in the language.

We mention possible directions of future research. Firstly, in this paper we study neither subject reduction nor strong normalization for $\rightarrow_\sigma$ and $\rightarrow_{\beta\sigma}$ and this is something we plan to do in the immediate future. Subject Reduction is not so easy to prove since for the case of σ-reduction we need to prove Strengthening. This study is necessary to design a syntax-directed system and a type checking algorithm for the Theory of Specifications. Secondly, this paper presents only a particular PTS (the one with specification TS) extended with pairs. It will be interesting to see if we could give a formulation of Pure Type Systems extended with pairs that includes the system in [SS01] and the one in the present paper.

Acknowledgement. We would like to thank the referees for their useful comments on an earlier version of this paper.

References

[Bar92] H. P. Barendregt. Lambda Calculi with Types. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 118–310. Oxford University Press, 1992.

[Bar99] Barras et al. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 1999.

[BM90] 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, Antibes*, pages 113,121, 1990.

[Luo89] Z. Luo. ECC, an extended calculus of constructions. In *4th. Symposium on Logic in Computer Science.*, 1989.

[PM89] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Prog ramming Languages*, Austin, January 1989. ACM.

[Pol94] E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, October 1994.

[SS01] 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.

[Sza97] N. Szasz. *A Theory of Specifications, Programs and Proofs.* PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden, June 1997.

[Wer94] B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Universite Paris 7, Paris, France, May 1994.