An operational approach to program extraction in the Calculus of Constructions

Maribel Fernández[1] and Paula Severi[2] *

[1] DI-LIENS, École Normale Supérieure/CNRS UMR 8548, 45 rue d'Ulm, 75005 Paris, France (maribel@di.ens.fr)
[2] Departimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy (severi@di.unito.it)

Abstract. The Theory of Specifications is an extension of the Calculus of Constructions where the specification of a problem, the derivation of a program, and its correctness proof, can all be done within the same formalism. An operational semantics describes the process of extracting a program from a proof of its specification. This has several advantages: from the user's point of view, it simplifies the task of developing correct programs, since it is sufficient to know just one system in order to be able to specify, develop and prove the correction of a program; from the implementation point of view, the fact that the extraction procedure is part of the system allows to control in a finer way its interactions with the rest of the system. In this paper we continue the study of the Theory of Specifications and propose a solution to restore subject reduction and strong normalization. Counterexamples for subject reduction and strong normalization for this theory have been shown in [RS02].

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 $f : A \to B$ with the property 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 the Theory of Specifications introduced in [SS01,SS02], 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 other words, the operational semantics describes explicitly the extraction procedure. This has several advantages: on the one hand, the

* 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.

extracted program, its specification and its correctness proof are all expressed in the same language; on the other hand, by making the extraction procedure explicit, we are able to control it in a finer way (this is analogous to what happens in explicit substitution calculi, where by making the substitution operation part of the system we are able to control its application).

The Theory of Specifications can also be used to model the method of program development by stepwise refinement (see for instance [Luo93] for a presentation of data refinement using type theory). The powerful type system of the Theory of Specifications allows us to define functions between specifications, and the σ-reduction can be used to compute the correctness condition of a refinement step.

The formulation of the Theory of Specifications given in [SS01] is based on Martin Löf's type theory and the formulation in [SS02] is based on the Calculus of Constructions. In this paper we focus on the second formulation. We add a pair constructor and a σ-conversion rule to the Calculus of Constructions. In our notation, we use the same pair constructor for types and objects. A Σ-type is abbreviated as a pair, i.e. $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,, \lambda x{:}A.P\rangle$. Due to the addition of the σ-reduction rules, the pair constructor becomes stronger than the strong-Σ [Luo89]: It has at least the expressive power of strong-Σ since it is possible to code the first and second projections and hence it is not necessary to add them as primitives [SS02]. Moreover, it goes beyond strong-Σ since 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 $\Pi x{:}A.(P\,x\,(f\,x))$.

The correctness of the program extraction process is proved in [SS01,SS02], by showing that the σ-reduction relation is normalizing and confluent, and that normal forms of specifications are pairs consisting of a program and its correctness proof. The components of a pair are typable in a subsystem of the Theory of Specifications called Verification Calculus, which excludes the formation of data types depending on propositions.

In a recent paper [RS02] it is shown that subject reduction and strong normalization in the Theory of Specifications of [SS02] do not hold. In this paper, we give a new formulation of the Theory of Specifications that enjoys the properties of subject reduction and strong normalization. These are important properties from the point of view of the implementation of the theory since they ensure that any evaluation strategy will eventually reach a normal form and there is no need for dynamic type checking.

Related Work. In Coq until version 6.3 [Bar99] the extraction procedure has been performed by means of an external function based on realizability interpretations [PM89b,PM89a]. The normal form of our σ-reduction corresponds to the functions $\mathcal{E}$ and $\mathcal{R}$ of [PM89b,PM89a] 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 [SS02], and it allows us to study the intermediate steps in the process of program extraction since an inhabitant s of a specification may need 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 [Tea] is also based on a reduction relation, however, there is an important difference: in all versions of Coq so far the extracted program is given in ML whereas the Theory of Specifications unifies the programming language and the logic (they are the same language). The σ-reduction is integrated into the system as part of its evaluation mechanism.

The Theory of Specifications is also related to other programming logics that allow the derivation of implementations as pairs program-proof developed in parallel. Among these systems, we can mention for instance the Deliverables [BM90] and the programming logic $\lambda\omega_L$ [Pol94].

In the Theory of Deliverables, 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 the Theory of Specifications, first and second order deliverables can be uniformly expressed using the Π-constructor as functions between specifications.

Poll's $\lambda\omega_L$ is a subsystem of the Verification Calculus. However, the notions of specification and implementation are not completely formalized in this language. Our notion of pair formalizes Poll's idea of couple-derivation rules which belongs to the meta-language in [Pol94].

In [Luo93] a different approach to program derivation is presented using an extended Calculus of Constructions. The idea is to specify a data type and then refine it until an implementation is obtained. This approach seems closer in spirit to the formal derivation of a program from a specification using the B method [Abr96]. The refinement function, which is defined externally in the theory of [Luo93] (it is not part of the reduction relation), can also be internalized in the Theory of Specifications, using functions between specifications.

Overview. We define an extended Calculus of Constructions and the Verification Calculus as Pure Type Systems in Section 2. In Section 3 we present a new version of the Theory of Specifications and give an example of program extraction. In Section 4 we prove several properties of the Theory of Specifications, in particular strong normalization and subject reduction, which are the main contributions of the paper. We conclude in Section 5.

2 Background

In this section we recall the definition of the Calculus of Constructions extended with an infinite type hierarchy [Luo89,Bar99] and introduce the Verification Calculus. We assume the reader to be familiar with the notion of Pure Type System [Bar92], and recall just the typing rules.
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 $\mathcal{A}$ of axioms and a set $\mathcal{R}$ of rules. A Pure type System 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
$$\frac{\Gamma \vdash U{:}k \quad \Gamma \vdash v{:}V}{\Gamma, x{:}U \vdash v{:}V}\ x\ \Gamma\text{-fresh}$$

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

Fig. 1. Typing rules for Pure Type Systems

The specification C for the extended calculus of constructions. The extended Calculus of Constructions [Luo89] is obtained as a Pure Type System with the following specification C.

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

Duplicating C 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 C, one for data types and the other for propositions.

1. Sorts. The sorts are either data-sorts $dType^i$ or prop-sorts $pType^i$ for $i \in \mathsf{Nat}$. Note that the sorts in VC are obtained by concatenating the letter d or p to the sorts of C. 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 C we add two axioms in the Verification Calculus:
$$dk : dk' \quad pk : pk'.$$

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

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

3 The Theory of Specifications

In this section we define the Theory of Specifications as a variant of [SS02] and give an example of an application. The reduction relation in this theory is the combination of β and σ. Counterexamples to subject reduction and strong normalization for β-reduction in the Theory of Specifications of [SS02] are shown in [RS02]. We restore subject reduction and strong normalization by restricting β-reduction to σ-normal forms.

3.1 Triplicating C for the Theory of Specifications

The Theory of Specifications is a Pure Type System extended with pairs. Its specification TS (sorts, axioms and product rules) 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 $i{\in}\mathsf{Nat}$. Note that for each sort k in C, 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 C we add three axioms in the Theory of Specifications:

$$dk : dk' \quad pk : pk' \quad sk : sk'$$

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

$$(uk_1, vk_2, vk_3)$$

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

3.2 Splitting the syntax

The set of pseudoterms is extended with expressions of the form $\langle u\,, v\rangle$ for pairs. 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 xd, yd, prop-variables by xp, yp and spec-variables by xs, ys.
The *heart* of a pseudoterm is defined as follows: $\mathsf{heart}(u) = u$ if u is either a variable, a sort or a pair, $\mathsf{heart}(\Pi x{:}U.V) = \mathsf{heart}(\lambda x{:}U.V) = \mathsf{heart}(V\,U) = \mathsf{heart}(V)$.
The set of pseudoterms is split in three sets:
A *data-pseudoterm* is a pseudoterm whose heart is a data-variable or a data-sort. We denote data-pseudoterms by A, B, $a, b \ldots$.
A *prop-pseudoterm* is a pseudoterm whose heart is a prop-variable or a prop-sort. We denote prop-pseudoterms by P, Q, $p, q, \ldots$

A *spec-pseudoterm* is a pseudoterm whose heart is a spec-variable, a spec-sort or a pair. 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: 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.

3.3 Operational semantics for program extraction

We define the reductions $\rightarrow_\sigma$ and $\rightarrow_\beta$ on pseudoterms. The first reduction, defined in [SS02], 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 an injective function Ψ such that $\Psi(xs) = \langle xd\,, xp\rangle$ for all spec-variables xs. The definition of $\rightarrow_\sigma$ is shown in Figure 2.

Splitting

$xs \rightarrow_\sigma \langle xd\,, xp\rangle$

$sType^n \rightarrow_\sigma \langle dType^n\,, \lambda xd{:}dType^n.(xd \rightarrow pType^n)\rangle$

Eliminating proofs from programs

$\Pi xp{:}P.A \rightarrow_\sigma A$ if $xp, xs \notin FV(A)$

$\lambda xp{:}P.a \rightarrow_\sigma a$ if $xp, xs \notin FV(a)$

$a\,p \rightarrow_\sigma a$

Curryfication

$\Pi xs{:}\langle A\,, P\rangle.U \rightarrow_\sigma \Pi xd{:}A.\Pi xp{:}(P\,xd).U$

$\lambda xs{:}\langle A\,, P\rangle.u \rightarrow_\sigma \lambda xd{:}A.\lambda xp{:}(P\,xd).u$

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

Distributivity

$\Pi x{:}U.\langle A\,, P\rangle \rightarrow_\sigma \langle \Pi x{:}U.A\,, \lambda f{:}\Pi x{:}U.A.\Pi x{:}U.(P\,(f\,x))\rangle$

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

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

Fig. 2. Definition of σ-reduction

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

[1] See [SS02] where these restrictions are imposed by a grammar.

To avoid the problems mentioned in [RS02] we restrict β-reduction to σ-normal forms.
We define the reduction $\to_\beta$ as the least relation on σ-normal forms that contains the following rules and is closed under the compatibility rules:

$$(\lambda xd{:}A.u)\, a \to_\beta u[a/xd]$$
$$(\lambda xp{:}P.u)\, p \to_\beta u[p/xp]$$

The reduction relation $\to_{\beta\sigma}$ on pseudoterms is defined as $\to_\sigma \cup \to_\beta$ and $\twoheadrightarrow_{\beta\sigma}$ denotes the transtive closure of $\to_{\beta\sigma}$.

Remark 1. The rule $xs \to_\sigma \langle xd\,, xp\rangle$ deserves more explanations: the left-hand side is a variable of the Theory of Specifications (a spec-pseudoterm) which we rewrite to the pair of variables $\Psi(xs) = \langle xd\,, xp\rangle$. It is therefore a rule scheme, representing an infinite number of rules, one for each xs. The variables xs of the Theory of Specifications are seen as constants in the signature of the rewrite system.

The following notion of *completeness* is needed to restrict substitution (though we restrict β-reduction to σ-normal forms, substitution is performed in the Application rule). We also need a modified definition of *freshness* with respect to a typing context.

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

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

3.4 Typing pairs

The Theory of Specifications is inductively defined by adding the rules in Figure 3 to the rules of Pure Type Systems when the specification is TS (see [SS02]).

The Theory of Specifications enjoys the following basic properties.

Lemma 3.1. (Classification)
If $\Gamma \vdash u{:}U$ then u, U are both data-pseudoterms, or both prop-pseudoterms, or both spec-pseudoterms.

Lemma 3.2. (Correctness)
If $\Gamma \vdash u{:}U$ then $\Gamma \vdash U{:}k$.

Theorem 3.3. (Interchange)
$\Gamma, xs{:}\langle A\,, P\rangle, \Delta \vdash u{:}U$ if and only if $\Gamma, xd{:}A, xp{:}(P\; xd), \Delta \vdash u{:}U$.

Pair Type
$$\frac{\Gamma \vdash A{:}dType^n \quad \Gamma \vdash P{:}A \to pType^n}{\Gamma \vdash \langle A\,,P\rangle{:}sType^n}$$

Pair Object
$$\frac{\Gamma \vdash a{:}A \quad \Gamma \vdash p{:}P\,a \quad \Gamma \vdash \langle A\,,P\rangle{:}sType^n}{\Gamma \vdash \langle a\,,p\rangle{:}\langle A\,,P\rangle}$$

Spec-variable
$$\frac{\Gamma \vdash xd{:}A \quad \Gamma \vdash xp{:}(P\,xd) \quad \Gamma \vdash \langle A\,,P\rangle{:}sType^n}{\Gamma \vdash xs{:}\langle A\,,P\rangle}\; xs \text{ is not in } \Gamma$$

Data-variable
$$\frac{\Gamma \vdash xs{:}\langle A\,,P\rangle}{\Gamma \vdash xd{:}A}\; xd \text{ is not in } \Gamma$$

Prop-variable
$$\frac{\Gamma \vdash xs{:}\langle A\,,P\rangle}{\Gamma \vdash xp{:}P\,xd}\; xp \text{ is not in } \Gamma$$

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

Fig. 3. Typing rules for pairs

This is proved by induction on the derivation. The only interesting case is the start rule, where we apply one of the rules for spec-variables, data-variables or prop-variables.

Theorem 3.4. (Inversion)

1. If $\Gamma \vdash \langle a\,,p\rangle{:}U$ then $\exists A, P$ such that $\Gamma \vdash a{:}A$, $\Gamma \vdash p{:}Pa$ and $V =_\sigma \langle A\,,P\rangle$, where $\Gamma \vdash A{:}dType^n$, $\Gamma \vdash P{:}A \to pType^n$, $\Gamma \vdash \langle A\,,P\rangle{:}sType^n$.
2. If $\Gamma \vdash \Pi x{:}U.V{:}k$ then $\Gamma, x{:}U \vdash V{:}k$ and $\Gamma \vdash U{:}k'$, where $(k', k, k) \in \mathcal{R}$.

The proof is by induction on the derivation.

3.5 An example of program extraction

In order to develop an example of program extraction, we have first to illustrate how σ-reduction is extended to deal with natural numbers. This extension is presented in [SS02] in a general setting for the Calculus of Inductive Constructions. We recall an example.
The inductive data type Nat with constructors zero and suc has the following elimination rule:

$$(\mathsf{natrec})\ \frac{\begin{array}{l}\Gamma \vdash n : \mathsf{Nat} \\ \Gamma, x{:}\mathsf{Nat} \vdash U : k,\ k \in \{dType^0, pType^0, sType^0\} \\ \Gamma \vdash u_1 : U[n/x] \\ \Gamma \vdash u_2 : \Pi m{:}\mathsf{Nat}.(U[m/x] \to U[\mathsf{suc}\, m/x])\end{array}}{\Gamma \vdash (\mathsf{natrec}\ u_1\ u_2\ n) : U[n/x]}$$

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 rule. 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. We consider the specification stating that every natural number not equal to zero has a predecessor. Using the definition $\Sigma x{:}A.P =^{\mathsf{def}} \langle A \,, P\rangle : sType^0$, this specification is written 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 σ-reduction, s is reduced to a pair. The first component is the extracted program, that is, the predecessor function, and the second component is the proof of its correctness.

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

4 Properties

In this section we prove some important meta-theoretical properties of the Theory of Specifications. Confluence has already been proved in [SS01]. Our main contribution is to prove that the formulation of the Theory of Specifications presented in this paper satisfies subject reduction and strong normalization.

Theorem 4.1. (Strong Normalization for σ)
The reduction relation $\rightarrow_\sigma$ is strongly normalizing.

This is proved by giving an interpretation function h that decreases with the reduction $\rightarrow_\sigma$, i.e. if $u \rightarrow_\sigma u'$ then $\mathsf{h}(u) > \mathsf{h}(u')$. See Definition A.1 in the appendix. We now give a proof of confluence of σ-reduction which is slightly different from the one presented in [SS01,SS02]. We use the following characterization of σ-normal forms.

Theorem 4.2. (Characterization of σ-normal forms)
Let a be a data-pseudoterm, p be a prop-pseudoterm and s be a spec-pseudoterm. Then a, p and s are in σ-normal form if and only if they can be defined by the following grammar:

$$A := xd \mid dk \mid \lambda xd{:}A.A \mid A\,A \mid \Pi xd{:}A.A$$
$$P := xp \mid pk \mid \lambda xd{:}A.P \mid \lambda xp{:}P.P \mid P\,A \mid P\,P \mid \Pi xd{:}A.P \mid \Pi xp{:}P.P$$
$$S := \langle A\,, P\rangle$$

Corollary 4.3.
The σ-normal forms are closed under β-reduction.

Theorem 4.4. (Confluence of σ and $\beta\sigma$)

1. $\rightarrow_\sigma$ is locally confluent.
2. $\rightarrow_\sigma$ is confluent.
3. $\rightarrow_{\beta\sigma}$ is confluent.

Proof: The first part is proved by inspecting the three critical pairs generated by the rules in the definition of σ-reduction. The second part follows by the previous lemma and strong normalization of $\rightarrow_\sigma$, using Newman's Lemma [New42]. The third part is a consequence of the confluence of σ (previous part) and β [SS01] using Corollary 4.3. Due to our restriction of β-reduction, in a $\beta\sigma$-reduction sequence the σ-steps are always performed before the β-steps. $\diamond$

The unique σ-normal form of a term u is denoted by $[\![u]\!]$. Note that the σ-normal form of s is always of the form $\langle a\,, p\rangle$ where a is a data-pseudoterm that does not contain prop-pseudoterms and p is a prop-pseudoterm.
The following theorem says that $[\![\ ]\!]$ is a "mapping" from the Theory of Specifications to the Verification Calculus. Derivation in the Verification Calculus is denoted by $\vdash_{\mathsf{VC}}$.

Theorem 4.6. (Soundness) [SS02]
Let $\Gamma \vdash u{:}U$.

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

The proof is by induction on the derivation.

Since $[\![\;]\!]$ computes the normal form of $\rightarrow_{\sigma}$, and the Verification Calculus is β-normalizing, we have that the Theory of Specifications is $\rightarrow_{\beta\sigma}$-normalizing [SS02]. Since the function $[\![\;]\!]$ is the identity for terms of the Verification Calculus, we have that the Theory of Specifications is conservative with respect to the Verification Calculus [SS02].
Using soundness and unicity of types in the Verification Calculus it is easy to prove unicity of types in the Theory of Specifications.

Lemma 4.7. (Unicity of Types) Let u be a term such that $\Gamma \vdash u{:}U$ and $\Gamma \vdash u{:}U'$.

1. If u is a data- or a prop-pseudoterm, then $U =_{\beta\sigma} U'$.
2. If u is a spec-pseudoterm, then $u =_{\beta\sigma} \langle a\,, p\rangle$, $U =_{\beta\sigma} \langle A\,, P\rangle$, and $U' =_{\beta\sigma} \langle A\,, P'\rangle$ such that $Pa =_{\beta\sigma} P'a$.

Corollary 4.8. (Unicity of Sorts) If $\Gamma \vdash U{:}k$ and $\Gamma \vdash U{:}k'$ then $k = k'$.

Theorem 4.9. (Strong Normalization for $\beta\sigma$)
The Theory of Specifications is $\beta\sigma$-strongly normalizing.

Proof: Suppose u is typable in the Theory of Specifications and there exists an infinite $\beta\sigma$-reduction sequence starting from u:

$$u = u_0 \rightarrow_{\beta\sigma} u_1 \rightarrow_{\beta\sigma} u_2 \ldots$$

Since σ is strongly normalizing, this reduction sequence should contain infinite β-steps. We consider the first term u_n in the sequence from which we perform a β-step. This step is performed on a σ-normal form due to our restriction of β-reduction. All the consecutive terms to u_n are in σ-normal forms because β-reduction does not create σ-redexes (Corollary 4.3). Hence we have an infinite β-reduction sequence starting from u_n. By soundness, either u_n is typable in the Verification Calculus or it is a pair whose components are typable in the Verification Calculus. This contradicts the fact that the Verification Calculus is β-strongly normalizing [Luo89]. $\diamond$

The following lemma is needed in the proof of subject reduction for β.

Lemma 4.11. (Expansion of contexts)
Let Γ' be a valid context (i.e. there are v, V such that $\Gamma' \vdash v{:}V$).
If $\Gamma \vdash u{:}U$ and $\Gamma' \twoheadrightarrow_{\beta\sigma} \Gamma$ then $\Gamma' \vdash u{:}U$.

We prove the statement for only one step of β or σ-expansion by induction on the derivation.

Theorem 4.12. (Subject Reduction for β)
If $u \rightarrow_{\beta} u'$ and $\Gamma \vdash u{:}U$ then $\Gamma \vdash u'{:}U$.

Proof: The term u is in σ-normal form. By soundness and subject reduction for β in the Verification Calculus, we have that

1. If u is a data or prop-pseudoterm then $[\![\Gamma]\!] \vdash_{\mathsf{VC}} [\![u']\!]{:}[\![U]\!]$.
2. If u is a spec-pseudoterm then $[\![U]\!] = \langle A\,, P\rangle$, $[\![u]\!] = \langle a\,, p\rangle$, $[\![u']\!] = \langle a'\,, p'\rangle$, $[\![\Gamma]\!] \vdash_{\mathsf{VC}} a'{:}A$ and $[\![\Gamma]\!] \vdash_{\mathsf{VC}} p'{:}P\,a$, and therefore $[\![\Gamma]\!] \vdash_{\mathsf{VC}} u'{:}[\![U]\!]$.

Since the context Γ and the type U are correct, we have that $\Gamma \vdash u'{:}[\![U]\!]$ (by Lemma 4.11). By applying σ-conversion rule, we have $\Gamma \vdash u'{:}U$. $\diamond$

From the equations $P \to A =_\sigma A =_\sigma (P \to Q) \to A$ we can type the term $(\lambda xp{:}P.xd\,(yp\,xp))\,yp$ in the context $yp : (P \to Q)$. This term is not in σ-normal form and it cannot be β-reduced. [2]
In order to prove that the rules that eliminate proofs from programs preserve the typing, we need to prove that the following rule is admissible:

$$\text{(Strengthening)}\ \frac{\Gamma, xp{:}P, \Delta \vdash u{:}U}{\Gamma, \Delta \vdash u{:}U} \quad xp, xs \notin FV(\Delta) \cup FV(u) \cup FV(U)$$

The following statement implies the admissibility of the Strengthening rule.

Lemma 4.14. (Strengthening)

1. Let U be in σ-normal form. If $\Gamma, xp{:}P, \Delta \vdash U{:}k$ and $xp, xs \notin FV(\Delta) \cup FV(U)$ then $\Gamma, \Delta \vdash U{:}k$.
2. If $\Gamma, xp{:}P, \Delta \vdash u{:}U$ and $xp, xs \notin FV(\Delta) \cup FV(u)$ then $\Gamma, \Delta \vdash u{:}\mathsf{nf}_{\beta\sigma}(U)$.

The $\mathsf{nf}_{\beta\sigma}U$ denotes the $\beta\sigma$-normal form of U.

Proof: The first part is proved using soundness, strengthening in the Verification Calculus and expansion of contexts (Lemma 4.11). The second part is proved by induction on the derivation. The interesting cases are the abstraction rule and the application rule. In both cases we use the previous part, unicity of sorts, and inversion. $\diamond$

Theorem 4.16. (Subject Reduction for σ)
If $u \to_\sigma u'$ and $\Gamma \vdash u{:}U$ then $\Gamma \vdash u'{:}U$.

Proof: This is proved by induction on the structure of u. The only interesting case is when u is itself a σ-redex. We distinguish cases according to the σ-rule applied. We show some cases. In each case we proceed by induction on the type derivation for u, distinguishing cases according to the last rule applied.

1. $u \equiv xs \to_\sigma \langle xd\,, xp\rangle \equiv u'$.
   Start Rule. We derive the same type for u' using Data-variable, Prop-variable and Pair Object.
   Spec-variable. We use Pair Object.
   All the other cases follow directly by induction.

[2] Otherwise it would β-reduce to a term containing the self-application $xd\,(yp\,yp)$ which is not typable (see [RS02] for a counterexample of strong normalization).

2. $u \equiv sType^n \rightarrow_\sigma \langle dType^n\,, \lambda xd{:}dType^n.(xd \rightarrow pType^n)\rangle \equiv u'$
   The only interesting case is the Axiom, all the other cases follow directly by induction.
   Axiom. Using Pair Type, it is sufficient to prove that $\Gamma \vdash dType^n{:}dType^{n+1}$ and $\Gamma \vdash \lambda xd{:}dType^n.(xd \rightarrow pType^n){:}dType^n \rightarrow pType^{n+1}$. The first is an axiom, and the second follows easily using the Abstraction rule.
3. $u \equiv \Pi xp{:}P.A \rightarrow_\sigma A \equiv u'$, if $xp, xs \notin A$.
   The only interesting case is Product, the others follow directly by induction.
   Product. The bound variable is a prop-variable, therefore we can eliminate the binder and use Strengthening.
4. $u \equiv \lambda xp{:}P.a \rightarrow_\sigma a \equiv u'$, if $xp, xs \notin A$.
   The only interesting case is Abstraction, the others follow directly by induction.
   Abstraction. We derive $\Gamma \vdash \lambda xp{:}P.a{:}\Pi xp{:}P.V \equiv U$ using $\Gamma, xp{:}P \vdash a{:}V$ and $\Gamma \vdash \Pi xp{:}P.V{:}k$. By soundness and σ-Conversion, $\Gamma, xp{:}P \vdash a{:}[\![V]\!]$. Using the characterization of σ-normal forms (see Theorem 4.2 in the appendix) and Strengthening we obtain $\Gamma \vdash a{:}[\![V]\!]$, and by σ-Conversion we derive $\Gamma \vdash a{:}U$.
5. $u \equiv a\,p \rightarrow_\sigma a \equiv u'$.
   The only interesting case is Application, the others follow directly by induction. The proof is similar to the previous case.
6. $u \equiv \Pi xs{:}\langle A\,, P\rangle.V \rightarrow_\sigma \Pi xd{:}A.\Pi xp{:}(P\,xd).V \equiv u'$.
   The only interesting case is Product, the others follow directly by induction.
   Product. We deduce $\Gamma \vdash \Pi xs{:}\langle A\,, P\rangle.V{:}U \equiv k_2$ from $\Gamma \vdash \langle A\,, P\rangle{:}k_1$ and $\Gamma, xs{:}\langle A\,, P\rangle \vdash V{:}k_2$. Using the Classification Lemma, $k_1 \equiv sType^n$, and we can derive $\Gamma \vdash A{:}dType^n$ and $\Gamma \vdash P{:}A \rightarrow pType^n$. We conclude using Product twice.
7. $u \equiv \lambda xs{:}\langle A\,, P\rangle.u \rightarrow_\sigma \lambda xd{:}A.\lambda xp{:}(P\,xd).u \equiv u'$.
   The only interesting case is Abstraction, the others follow directly by induction. We use the Interchange Theorem , and the previous part.
8. $u \equiv u\,\langle a\,, p\rangle \rightarrow_\sigma u\,a\,p \equiv u'$.
   The only interesting case is Application, the others follow directly by induction. Again, we use the Interchange Theorem and the following property: $V \twoheadrightarrow_\sigma V[\langle xd\,, xp\rangle/xs]$ using the σ-rule for spec-variables, for which we have already proved preservation of types. We conclude by using σ-Conversion and Application twice.
9. The Distributivity rules are proved by induction on the type derivation, using the Classification Lemma, Soundness, and Inversion.

$\diamond$

5 Further Work

A natural extension of the Theory of Specifications would be the introduction of explicit substitutions and explicit control of ressources (copying and erasing). We plan to define a version of the Theory of Specifications with explicit substitutions and ressource management, and study the interactions between β, σ and

substitution in this framework. We hope that this would allow us to define an efficient strategy of reduction for the Theory of Specifications.
It is necessary to define a syntax directed set of rules for the Theory of Specifications for the type inference algorithm. We should remove rules like $\beta\sigma$-conversion which introduce ambiguities in the derivation. Proving the equivalence between the syntax-directed set of rules and the Theory of Specifications is not an easy task.

References

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Bar92] 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.

[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*, pages 113–121, 1990.

[Luo89] Z. Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of LICS '89*, IEEE, pages 386–395. IEEE Computer Society Press, 1989.

[Luo93] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3:333–363, 1993.

[New42] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[PM89a] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[PM89b] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, 1989.

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

[RS02] F. van Raamsdonk and P. Severi. Eliminating proofs from programs. In *Proceedings of Third International Workshop on Logical Frameworks and Meta-Languages*, volume 70.2 of *Electronic Notes in Theoretical Computer Science*, 2002.

[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.

[SS02] P. Severi and N. Szasz. Internal Program Extraction in the Calculus of Inductive Constructions. In *6th Argentinian Workshop in Theoretical Computer Science (WAIT'02), 31st JAIIO*, 2002.

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

A Appendix

Definition A.1. (Interpretation function)
The function h from the set of pseudoterms into Nat is defined by induction as follows.

VARIABLES.
$\mathsf{h}(xd) = 0$
$\mathsf{h}(xp) = 0$
$\mathsf{h}(xs) = 1$

SORTS.
$\mathsf{h}(dType^i) = 0$
$\mathsf{h}(pType^i) = 0$
$\mathsf{h}(sType^i) = 1$

PAIRS.
$\mathsf{h}(\langle a\,, p\rangle) = \mathsf{h}(a) + \mathsf{h}(p)$

PRODUCTS.

$$\begin{aligned}
\mathsf{h}(\Pi xd{:}A.B) &= \mathsf{h}(A) + \mathsf{h}(B)\\
\mathsf{h}(\Pi xp{:}P.B) &= \mathsf{h}(P) + \mathsf{h}(B) + 1\\
\mathsf{h}(\Pi xs{:}S.B) &= \mathsf{h}(S) + \mathsf{h}(B) + 2\\
\mathsf{h}(\Pi xd{:}A.Q) &= \mathsf{h}(A) + \mathsf{h}(Q)\\
\mathsf{h}(\Pi xp{:}P.Q) &= \mathsf{h}(P) + \mathsf{h}(Q)\\
\mathsf{h}(\Pi xs{:}S.Q) &= \mathsf{h}(S) + \mathsf{h}(Q) + 1\\
\mathsf{h}(\Pi xd{:}A.T) &= 3 * \mathsf{h}(A) + 2 * \mathsf{h}(T) + 1\\
\mathsf{h}(\Pi xp{:}P.T) &= 3 * \mathsf{h}(P) + 2 * \mathsf{h}(T) + 4\\
\mathsf{h}(\Pi xs{:}S.T) &= 6 * \mathsf{h}(S) + 4 * \mathsf{h}(T) + 10
\end{aligned}$$

ABSTRACTIONS.

$$\begin{aligned}
\mathsf{h}(\lambda xd{:}A.a) &= \mathsf{h}(A) + \mathsf{h}(a)\\
\mathsf{h}(\lambda xp{:}P.a) &= \mathsf{h}(P) + \mathsf{h}(a) + 1\\
\mathsf{h}(\lambda xs{:}S.a) &= \mathsf{h}(S) + \mathsf{h}(a) + 2\\
\mathsf{h}(\lambda xd{:}A.p) &= \mathsf{h}(A) + \mathsf{h}(p)\\
\mathsf{h}(\lambda xp{:}P.p) &= \mathsf{h}(P) + \mathsf{h}(p)\\
\mathsf{h}(\lambda xs{:}S.p) &= \mathsf{h}(S) + \mathsf{h}(p) + 1\\
\mathsf{h}(\lambda xd{:}A.s) &= 2 * \mathsf{h}(A) + \mathsf{h}(s) + 1\\
\mathsf{h}(\lambda xp{:}P.s) &= 2 * \mathsf{h}(P) + \mathsf{h}(s) + 2\\
\mathsf{h}(\lambda xs{:}S.s) &= 2 * \mathsf{h}(S) + \mathsf{h}(s) + 4
\end{aligned}$$

APPLICATIONS.

$$\begin{aligned}
\mathsf{h}(a\,b) &= \mathsf{h}(a) + \mathsf{h}(b)\\
\mathsf{h}(a\,p) &= \mathsf{h}(a) + \mathsf{h}(p) + 1\\
\mathsf{h}(a\,s) &= \mathsf{h}(a) + \mathsf{h}(s) + 2\\
\mathsf{h}(p\,a) &= \mathsf{h}(p) + \mathsf{h}(a)\\
\mathsf{h}(p\,q) &= \mathsf{h}(p) + \mathsf{h}(q)\\
\mathsf{h}(p\,s) &= \mathsf{h}(p) + \mathsf{h}(s) + 1\\
\mathsf{h}(s\,a) &= 2 * \mathsf{h}(a) + \mathsf{h}(s) + 1\\
\mathsf{h}(s\,p) &= 2 * \mathsf{h}(p) + \mathsf{h}(s) + 2\\
\mathsf{h}(s\,t) &= 2 * \mathsf{h}(t) + \mathsf{h}(s) + 4
\end{aligned}$$