Verifying Properties of Module Construction in Type Theory*

Bernhard Reus Thomas Streicher

Institut für Informatik, Universität München,
Leopoldstraße 11B, D-8000 München 40 (Germany)
`[reus|streiche]@informatik.uni-muenchen.de`

Abstract. This paper presents a comparison between algebraic specifications-in-the-large and a type theoretical formulation of modular specifications, called *deliverables*. It is shown that the laws of module algebra can be translated to laws about deliverables which can be proved correct in type theory. The adequacy of the *Extended Calculus of Constructions* as a possible implementation of type theory is discussed and it is explained how the reformulation of the laws is influenced by this choice.

1 Introduction

One of the most popular approaches to formal specification of software systems, i.e. *programs* or *program modules*, is the theory of algebraic specifications. This approach is suitable for *programming-in-the-small*, i.e. the specification of *one single* (program) module. For building bigger structured software systems – the discipline of so-called *programming-in-the-large* – it is necessary to deal with modules and their specifications as objects, because one wants to define functions and predicates on modules and their specifications. Admittedly in specification languages such as *CLEAR* [Burstall, Goguen 80] or *ASL* [Wirsing 86] one has certain *built-in linguistic constructs* for forming compound specifications but they *cannot be considered as functions operating on specifications* as there is not something like a "sort or type of specifications". ([Leszczylowski, Wirsing 91] is a proposal in this direction.) Furthermore one has no linguistic means to refer to modules as objects but only can formulate predicates on modules, i.e. specifications.

It seems to be fairly difficult to extend the formalism of algebraic specifications in a simple way to the needs of programming-in-the-large. One reason is that types of modules cannot be expressed without employing the concepts of *dependent types* and *sums of families of types* whereas the type systems used in algebraic specifications do not go beyond simply typed λ-calculus. The other reason is that in the theory of algebraic specifications a specification does not simply denote a class of modules but also contains some syntactic information (e.g. about the underlying signature or the special form of the axioms in order to

* This work was partially sponsored by the BMFT-project KORSO
Accepted for MFCS'93, Gdańsk, Poland

guarantee properties such as confluence or termination). Therefore a drastic shift of concepts has to take place in order to adapt specifications to the needs of programming-in-the-large. A very appealing suggestion into this direction has been made and forcefully argued for by R. Burstall and J. McKinna when they introduced their concept of *deliverables* [Burstall, McKinna 92] [McKinna 92].
We will show that certain relations between specifications as formulated by the laws of so called *module algebra* [Bergstra et al. 89] can be reformulated in the framework of deliverables. Thereby we heavily build on previous work by Zhaohui Luo, see [Luo 91], where he has expressed the fundamental operations on specifications as operations on deliverables. The version of type theory we have chosen is the *Extended Calculus of Constructions* (XCC) [Luo 89] as it provides an infinite type hierarchy with sums and it is implemented within the LEGO system [Luo et al. 92]. Having a good implementation available is of great importance as machine assistance is indispensable for constructing and checking completely formalized proofs.
The paper is organized as follows: Section 2 introduces the concept of a *deliverable* as a generalization of a specification-in-the-small, whereas Section 3 discusses them in-the-large and provides also their algebraic counterparts. The main contribution of this paper is contained in Section 4 where the module algebra axioms and their type theoretic analagons are presented. A proof sketch for one of the laws is included in order to illustrate the typical problems arising from the choice of XCC. Finally Section 5 contains a short summary.

2 Specifications as deliverables

In this section we recap the basic intuition behind the *generalization from algebraic specifications to deliverables* [Luo 91]. Essentially there is nothing new in our presentation maybe with the exception that we put more emphasis on explaining the differences between algebraic specifications and their generalization to deliverables.
In the theory of algebraic specifications a specification is given by two components [Wirsing 89]. A signature indicating the *type of modules under consideration* and a collection of axioms expressing which *laws the intended subcollection of this type of modules should satisfy*.
Now the *deliverable approach* starts from the idea of *model class semantics*. The basic assumption is that the collection of specified objects is more fundamental than the form of its syntactic presentation. Therefore a type of modules is not represented indirectly by a syntactic object like a signature but more generally by an *arbitrary type*. Typically the modules of a signature with one sort s and one unary function symbol $f : S \rightarrow S$ are represented by the type

$$(\sum S : Type)\ S \rightarrow S\ .$$

For the notation see e.g. [Nordström et al. 90]. Thus under presence of dependent types and sums of families of types there is no longer need for signatures as objects. Due to α-conversion for bound variables problems with renamings of signatures disappear.

Now a deliverable in the sense of Burstall and McKinna is a pair $\langle S, P\rangle$ where S is a type and $P : S \rightarrow\ Prop$ is a predicate on S. Accordingly, the deliverable approach amounts to manipulating and stating relations between the model classes corresponding to algebraic specifications in a constructive way.

An important aspect of deliverables is that they organize into a category *DELIV* whose underlying type of objects is

$$(\sum S : Type)\ S \rightarrow\ Prop\ .$$

The type $S \rightarrow Prop$ will be abbreviated by $Pred(S)$ throughout the paper. Note that since S itself can be a type we must avoid circular types. But absence of circularity is checked automatically by the implementation of XCC in the LEGO system.

A morphism from deliverable $D1 = \langle S_1, P_1\rangle$ to deliverable $D2 = \langle S_2, P_2\rangle$ is a function $f : S_1 \rightarrow S_2$ together with the proof that the proposition

$$(\Pi x : S_1)\ P_1(x) \rightarrow P_2(f(x)) \qquad (*)$$

holds. The most important aspect of deliverable morphisms is that they provide an adequate concept of refinement step as a morphism f from D_1 to D_2 can be interpreted as a *verified constructive implementation of* D_2 *in terms of* D_1. This notion of implementation is most adequate as it makes a clear distinction between the *computational part* of an implementation - the function $f : S_1 \rightarrow S_2$ and the proof of the *verification condition* (*) expressing that f transforms modules of type S_1 satisfying requirement P_1 into modules of type S_2 satisfying requirement P_2.

This suggests a methodology of program development by stepwise implementation – employing n-ary deliverable morphisms instead of only unary ones – where in each implementation step one already gets some part of the final functional program. Thus the deliverable approach provides an accurate mathematical setting for the *stepwise refinement method* for program development.

The module algebra axioms state equalities between specifications which can be used to flatten structural specifications, i.e. expand compound specifications to explicit form. Structured specifications are used in particular for building reusable specifications. As well known from the theory of algebraic specifications the operations *derive*, *translate*, *sum*, and *union* are not defined on the collection of *all* algebraic specifications but are inherently partial. This *partiality is made explicit by an appropriate typing* of these operators. Furthermore in the formulation of the laws of module algebra there is a certain amount of *informality* in expressing *side conditions of syntactic nature*, i.e. assumptions about signatures and signature morphisms. In the deliverable formulation these informalities are avoided by expressing these side conditions by typing assumptions.

3 Specification-in-the-large

We have already seen how specifications can be expressed as deliverables. But in this framework not only specifications-in-the-small can be represented adequately but also specifications-in-the-large. Most of the relevant specification operations have been already described in [Luo 91]. After a short summary of these operations we will take a closer look on the module algebra axioms of [Bergstra et al. 89] and check if they correspondingly hold for deliverables.
First, let us recall the basic notions of algebraic specifications of [Wirsing 89]: We assume the class *Sign* of all signatures to be given. The collection *Spec* of all specifications consists of all pairs $\langle \Sigma, C\rangle$ of a signature Σ together with a class C of Σ-algebras, which is, in general, closed under isomorphism (i.e. for all $A, B \in Alg(\Sigma)$, $A \in C$ and $A \cong B$ implies $B \in C$).

$$Spec =_{def} \{\langle \Sigma, C\rangle \mid \Sigma \in Sign \text{ and } C \subseteq Alg(\Sigma)\ \}.$$

On *Spec* we define two projection functions: the left projection

$$sig : Spec \to Sign,\ \ sig(\langle \Sigma, C\rangle) =_{def} \Sigma \text{ for all } \langle \Sigma, C\rangle \in Spec,$$

assigns to each element of *Spec* its signature, the right projection whereas

$$Mod : Spec \to \{C \subseteq Alg(\Sigma) \mid \Sigma \in Sign\},$$
$$Mod(\langle \Sigma, C\rangle) =_{def} C \text{ for all } \langle \Sigma, C\rangle \in Spec,$$

assigns to each element of *Spec* its class of models.
For any signature Σ and any set E of Σ-formulas, the basic specification $< \Sigma, E >$ denotes the pair $\langle \Sigma, Alg(\Sigma, E)\rangle$ in *Spec*, i.e.

$$sig(< \Sigma, E >) =_{def} \Sigma,$$
$$Mod(< \Sigma, E >) =_{def} \{A \in Alg(\Sigma) \mid A \models E\}.$$

Thus, the models of $< \Sigma, E >$ are all Σ-algebras which satisfy E.

3.1 Specification Operations

union The definition of $\cup$ for algebraic specifications requires that the signatures of both arguments are equal i.e. $sig(sp_1) = sig(sp_2)$:

$$sig(sp_1 \cup sp_2) =_{def} \Sigma,$$
$$Mod(sp_1 \cup sp_2) =_{def} Mod(sp_1) \cap Mod(sp_2).$$

The corresponding type theoretical definition is :

$$\mathit{union} : \Pi\ S{:}Type.\ Pred(S) \to Pred(S) \to Pred(S)$$
$$\mathit{union}\ (S)(SP_1)(SP_2) =_{def} \lambda\ x{:}S.\ SP_1(x) \wedge SP_2(x)$$

where, as mentioned in Section 2, *Pred* abbreviates $\lambda S{:}Type.\ S \to Prop$. The intersection of model classes corresponds to the conjunction of formulas that the module x must fulfil. The condition that both specifications must have the same signature is expressed by the type of *union*.

derive The ASL-operation '**derive from**' is the basis for renaming specifications and exporting subsignatures. For any signature Σ', any signature morphism $\sigma : \Sigma \to \Sigma'$ and any class C' of Σ'-algebras, it is defined as follows:

$$sig(\textit{derive from } \langle \Sigma', C' \rangle \textit{by } \sigma) =_{def} \Sigma \ ,$$
$$Mod(\textit{derive from } \langle \Sigma', C' \rangle \textit{by } \sigma) =_{def} \{A|_\sigma \in Alg(\Sigma) \mid A \in C'\} \ ,$$

where $A|_\sigma$ is the σ-reduct of A.
The type theoretic definition is:

$$\textit{derive} : \Pi\ S', S{:}Type.\ Pred(S') \to (S' \to S) \to Pred(S)$$
$$\textit{derive}\ (S')\ (S)\ (SP')\ (\varphi) =_{def} \lambda x{:}S.\ \exists s{:}S'.\ (SP'\ s) \wedge \varphi(s) = x$$

If we read $\{A|_\sigma \in Alg(\Sigma) \mid A \in Mod(SP')\}$ as $\{A \in Alg(\Sigma) \mid \exists B \in Alg(\Sigma').\ B \in Mod(SP') \wedge \sigma(B) = A\}$ then "$A \in Alg(\Sigma)$" corresponds to the typing "$x{:}S$", "$\exists B \in Alg(\Sigma')$" corresponds to "$\exists s{:}S'$" and "$B \in Mod(SP') \wedge \sigma(B) = A$" corresponds to "$(SP'\ s) \wedge \varphi(s) = x$".
Note the difference between the signature morphism σ and the induced structure map φ which goes in the opposite direction.

export The *export* operation can be defined in terms of *derive*:

$$\textit{export } \Sigma_0 \textit{ from } \langle \Sigma, C \rangle =_{def} \textit{derive from } \langle \Sigma, C \rangle \textit{by } i,$$

where i is the canonical injection $i : \Sigma_0 \to \Sigma$, $i(x) =_{def} x$.
In the deliverables approach we write for the first projection:

$$\textit{derive}\ (\textstyle\sum x{:}A.B(x))\ (A)\ (P)\ (\pi_1)$$

instead of the *export*.

translate The algebraic definition is:

$$sig(translate\langle \Sigma, C \rangle \textit{ with } \sigma) =_{def} \Sigma',$$
$$Mod(translate\langle \Sigma, C \rangle \textit{ with } \sigma) =_{def} \{A \in Alg(\Sigma') |\ A|_\sigma \in C\},$$

where $A|_\sigma$ is the Σ-reduct of A w.r.t. $\sigma : \Sigma \to \Sigma'$.
The type theoretic definition is:

$$\textit{translate} : \Pi\ S', S{:}Type.\ Pred(S) \to (S' \to S) \to Pred(S')$$
$$\textit{translate}(S')(S)(SP)(\varphi) =_{def} \lambda x{:}S'.\ SP(\varphi(s))$$

The condition "$A \in Alg(\Sigma')$" corresponds to the typing "$x{:}S'$" and "$A|_\sigma \in C$" corresponds to "$SP(\varphi(s))$". In algebraic specifications the function φ in *translate* and *derive* can only be generated by a signature morphism which is characterized by its ability to forget, identify, and rename. Since renaming is irrelevant due to α-conversion for simulating this approach one should use for φ only functions that project and identify (via λ-abstraction and products with identical components). In the type theoretic approach *translate* and *derive* are therefore stronger than their algebraic counterparts because φ is not necessarily induced by a signature morphism.

sum: putting two specifications together In algebraic specifications the $+$-operator is needed in order to build the amalgamation of two specifications, i.e. for gluing them together at a common interface. More formally, given specifications $sp_1, sp_2 \in Spec$ with signatures $sig(sp_1) = \Sigma_1$ and $sig(sp_2) = \Sigma_2$,

$$sp_1 + sp_2 =_{def} (translate\ sp_1\ with\ in_1) \cup (translate\ sp_2\ with\ in_2).$$

where $in_k : \Sigma_k \to \Sigma_1 \cup \Sigma_2$ embed the signatures into their union by $in_k(x) = x$ for $x \in \Sigma_k$, $k = 1, 2$.
The $\cup$ on signatures has no direct analogue on the module level, since in the deliverables approach there are no more signatures indicating the *shared parts*. Therefore the common interface must be given explicitly. This shared part must of course fulfil the requirements of both specifications. *sum* is the type theoretic counterpart of $+$. Note that S denotes the *shared part*.

$$\begin{aligned} &sum : \Pi S{:}Type.\Pi R,T{:}S \to Type.\ Pred(\Sigma s{:}S.R(s)) \to Pred(\Sigma s{:}S.T(s)) \to \\ &\qquad Pred(\Sigma s{:}S.\,(R(s) \times T(s))) \\ &sum\ S\ R\ T\ SP_1\ SP_2 =_{def} \lambda x{:}(\Sigma s{:}S.\,R(s) \times T(s)). \\ &\qquad SP_1\langle \pi_1(x), \pi_1(\pi_2(x))\rangle \wedge SP_2\langle \pi_1(x), \pi_2(\pi_2(x))\rangle \end{aligned}$$

Now "$\sum s{:}S.\,R(s) \times T(s)$" corresponds to "$\Sigma_1 \cup \Sigma_2$" "$S$" corresponds to "$\Sigma_1 \cap \Sigma_2$", "$R$" to "$\Sigma_1$", "$T$" to "$\Sigma_2$" and "$\lambda x{:} \sum s{:}S.\,R(s) \times T(s).\ (\pi_1(x), \pi_k(\pi_2(x)))$" is the projection function that corresponds to "in_k". *sum* combines two principles, namely adding axioms (union) and enlarging the signature or structure type and can therefore serve to derive all other notions of combining specifications like *enrich* or *parallel* [Luo 91].

3.2 Equality on deliverables on the same underlying type

Before equations on specifications as deliverables can be proved one has to fix an equality on them, i.e. on $Pred(S)$. Certainly we want to consider two deliverables as equal if the predicates are equivalent (and not if they are equal intensionally), i.e.:

$$\begin{aligned} &equal : \Pi\ S{:}Type.\ Pred(S) \to Pred(S) \to Prop \\ &equal\ (S)(D_1)(D_2) = \forall\ s{:}S.\ D_1(s) \Longleftrightarrow D_2(s) \end{aligned}$$

For the sake of readability we write $SP_1 =_S SP_2$ instead of *equal* $(S)(SP_1)(SP_2)$ and $=$ without decoration denotes Leibniz equality, i.e.
$(a = b : S) =_{def} \forall P{:}Pred(S).\ P(a) \Rightarrow P(b)$.

4 Module algebra axioms in type theory

4.1 The module algebra

Let us recall the Module Algebra specification of [Bergstra et al. 89] in the form of [Wirsing 89] omitting the type theoretically unnecessary operation *reach* (*reach* can be rephrased as a second order formulae expressing the induction principle). Therefore, the equations concerning *reach* (T2) and (S2) are left out and (E1) is omitted, because it corresponds to the definition of deliverables.

spec *BSA* $\equiv$
extend *SIGNATURE, FORMULA* **by**
sort *spec*
constructors $<.,.>$: *signature, set formula* $\rightarrow$ *spec*,
$. \cup .$: *spec, spec* $\rightarrow$ *spec*,
export . from . : *signature, spec* $\rightarrow$ *spec*,
translate . with . : *spec, signature morphism* $\rightarrow$ *spec*
functions *sig* : *spec* $\rightarrow$ *signature*
axioms $\forall$ sp, sp_1, sp_2 : *spec*, $\Sigma, \Sigma_1, \Sigma', \Sigma''$: *signature*,
E_1, E_2 : *set formula*, $\sigma : \Sigma \rightarrow \Sigma'$, $\sigma' : \Sigma' \rightarrow \Sigma''$: *signature morphism*.

(U1) $sp \cup sp = sp$,
(U2) $sp_1 \cup sp_2 = sp_2 \cup sp_1$,
(U3) $sp \cup (sp_1 \cup sp_2) = (sp \cup sp_1) \cup sp_2$,
(U4) $< \Sigma, E_1 > \cup < \Sigma, E_2 > = < \Sigma, E_1 \cup E_2 >$,

(E2) $\Sigma = sig(sp_1) \cap sig(sp_2) \Rightarrow$
$(export\ \Sigma\ from\ sp_1) \cup (export\ \Sigma\ from\ sp_2) = export\ \Sigma\ from\ (sp_1 + sp_2)$,
(E3) $\Sigma \subseteq \Sigma_1 \subseteq sig(sp) \Rightarrow$
$export\ \Sigma\ from(export\ \Sigma_1\ from\ sp) = export\ \Sigma\ from\ sp$,
(E4) $export\ sig(sp)\ from\ sp = sp$,

(T1) $translate < \Sigma, E > with\ \sigma = < \Sigma', \sigma^*(E) >$,
(T3) $translate\ (sp_1 \cup sp_2)\ with\ \sigma = (translate\ sp_1\ with\ \sigma) \cup (translate\ sp_2\ with\ \sigma)$,
(T4) $translate\ (export\ \Sigma\ from\ sp)\ with\ \sigma = export\ \Sigma'\ from\ (translate\ sp\ with\ \hat{\sigma})$,
(T5) $translate\ (translate\ sp\ with\ \sigma)\ with\ \sigma' = translate\ sp\ with\ \sigma' \circ \sigma$,

(αh) $\Sigma_1 \subseteq \Sigma = sig(sp) \wedge \sigma|_{\Sigma_1} = id \wedge injective(\sigma) \Rightarrow$
$export\ \Sigma_1\ from\ sp = export\ \Sigma_1\ from\ (translate\ sp\ with\ \sigma)$,
endspec

Here $\widehat{\sigma} : sig(sp) \rightarrow \Sigma' \cup (sig(sp) \setminus \Sigma)$ denotes the extension of σ to $sig(sp)$ defined by $\widehat{\sigma}(x) = \sigma(x)$, if $x \in \Sigma$, and x, otherwise. σ^* is the obvious extension of σ to formulas over $dom(\sigma)$.

4.2 The analogue of the laws in type theory

We will now present the corresponding formulation in a type theoretic framework. Therefore, we assume the following context:

$A,D,E : Type,\ \ B,C : A \rightarrow Type,$
$S_B =_{def} \sum x{:}A.B(x),\quad S_C =_{def} \sum x{:}A.C(x),\quad S_{AC} =_{def} \sum x{:}A.\ B(x) \times C(x),$
$F : S_B \rightarrow Type,\quad S_{BF} =_{def} \sum y{:}S_B.F(y).$

Let $\hat{P}$ be an abbreviation for $\lambda s{:}S_B.\ P\langle \pi_1(s), \pi_2(s) \rangle$ (analogously for Q) and use the following definitions of *comp* and *join*, where *comp* denotes function composition and *join* takes two functions and yields a function on the dependent sum of their domains, since the domain of the second function may depend on the domain of the first one.

$comp : \Pi\ A,B,C{:}Type.\ (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$

$comp\ A\ B\ C\ f\ g\ s =_{def} g(f(s))$
$comp_{\pi_1} =_{def} comp(S_{BF})(S_B)(A)(\pi_1)(\pi_1)$

$join : \Pi X,Y{:}Type.\ \Pi B{:}X\to Type.\ \Pi C{:}Y\to Type.\ \Pi f{:}(X\to Y).(\Pi x{:}X.B(x) \to C(f(x))$
$\quad \to (\sum x{:}X.B(x)) \to (\sum y{:}Y.C(y))$
$join\ X\ Y\ B\ C\ f\ g =_{def} \lambda z{:}(\sum x{:}X.B(x)).\ \langle\, f(\pi_1(z)),\ g(\pi_1(z))\,(\pi_2(z))\,\rangle$

Let Co denote this context of definitions. For the logical connectives we use the usual second-order definitions, see e.g. [Luo 91].

(U1) $\forall SP{:}Pred(A).\ (union\ A\ SP\ SP) =_A SP$

(U2) $\forall SP_1,SP_2 : Pred(A).\ (union\ A\ SP_1\ SP_2) =_A (union\ A\ SP_2\ SP_1)$

(U3) $\forall\ SP_1,SP_2,SP_3 : Pred(A).$
$(union\ A\ SP_1\ (union\ A\ SP_2\ SP_3)) =_A (union\ A\ (union\ A\ SP_1\ SP_2)\ SP_3)$

(E2) $\forall P{:}Pred(S_B).\forall R{:}Pred(S_C).$
$(union\ A\ (derive\ S_B\ A\ \hat{P}\ \pi_1)\ (derive\ S_C\ A\ \hat{R}\ \pi_1)) =_A$
$(derive\ S_{AC}\ A\ (sum\ A\ B\ C\ \hat{P}\ \hat{R})\ \pi_1)$

(E3) $\forall P{:}Pred(S_{BF}).$
$(derive\ S_B\ A\ (derive\ S_{BF}\ S_B\ P\ \pi_1)\ \pi_1) =_A (derive\ S_{BF}\ A\ P\ comp_{\pi_1})$

(E4) $\forall P{:}Pred(A).\ (derive\ A\ A\ P\ (\lambda s{:}A.s)) =_A P$

(T3) $\forall P,R{:}Pred(A).\ \forall f{:}D\to A.$
$(translate\ D\ A\ (union\ A\ P\ R)\ f) =_D (union\ D\ (translate\ D\ A\ P\ f)\ (translate\ D\ A\ R\ f))$

(T4) $\forall P{:}Pred(S_B).\ \forall f{:}D\to A.$
$(translate\ D\ A\ (derive\ S_B\ A\ P\ \pi_1)\ f\ ) =_D (derive\ (\sum x{:}D.B(f(x)))\ D$
$(translate\ \sum x{:}D.B(f(x))\ S_B\ \hat{P}\ (join\ f\ \lambda x{:}D.\lambda y{:}B(f(x)).y))\ \pi_1)$

(T5) $\forall P{:}Pred(A).\ \forall f{:}D\to A.\ \forall g{:}E\to D.$
$(translate\ E\ D\ (translate\ D\ A\ P\ f)\ g) =_E (translate\ E\ A\ P\ f\ (comp\ E\ D\ A\ g\ f))$

(αh) $\forall P{:}Pred(S_B).\ (\forall x{:}A.\ \forall y{:}B(x).\ P\langle x,y\rangle \Rightarrow \exists z{:}F\langle x,y\rangle.true\ ) \Rightarrow$
$(derive\ S_B\ A\ P\ \pi_1) =_A (derive\ S_{BF}\ A\ (translate\ S_{BF}\ S_B\ \hat{P}\ \pi_1)\ comp_{\pi_1})$

Remark: The axioms (U4) and (T1) reflect the definition of the operators *union* and *translate* as deliverables and are therefore omitted in the type theoretic formulation. Note that instead of quantifying over signature morphisms due to the generality of deliverables we have to quantify over functions between the underlying types (in the opposite direction).
The choice of the projection functions and the bracketing of the products is, of course, arbitrary. In this presentation we always chose the first projection, π_1, and the structure types are bracketed correspondingly, e.g. $\sum x{:}A.B(x)$. But the projected type may be only a part of A which is independent from the other components of the (sum) type. The projection would then yield some arbitrary

components of the dependent sum. Still it can be written as composition of first and second projection functions. As a consequence, the stated equations are valid up to canonical isomorphism between product types.
(E2)–(E4): The *export* operation used in *BSA* must be expressed via *derive* using a projection function (cf. Section 3.1) Therefore, conditions on the signatures (formulated via *sig* in *BSA*) have to be expressed by the appropriate typing condition on the level of deliverables. In (E2) the information "$\Sigma = sig(sp_1) \cap sig(sp_2)$" is expressed by the fact that sp_1 has structure type $\sum x{:}A.B(x)$ and sp_2 has type $\sum x{:}A.C(x)$ (with B, C arbitrary families of structure types). Since Σ corresponds to A, which is the common part, the antecedent is modelled adequately.
Note that using $\hat{P}$ and $\hat{R}$ in formulating the liftings of (E2),(T4) and (αh) has become necessary in order to prove them in XCC where one cannot prove

$$P(s) \Leftrightarrow P(\langle \pi_1(s), \pi_2(s) \rangle)$$

uniformly in P and s. In other words the Leibniz equality of s and $\langle \pi_1(s), \pi_2(s) \rangle$ is not provable in XCC. If we would use type theory with an eliminator for $\sum$-types à la Martin-Löf, cf. [Nordström et al. 90], then the invariance of predicates under surjective pairing would become provable and we could avoid the tricky use of $\hat{P}$. In the next Section we will give a proof sketch for (E2) and explain this in more detail.
(T3) and (T5) are obvious. (T4) can avoid the $\hat{\sigma}$ construction of *BSA* using a concrete function on structures which is built via *join*.
When lifting (αh) the first part of the premiss expressing the subsignature assumption is again formulated by assuming that the type under consideration is a $\sum$-type of the appropriate form. The assumption of injectivity of σ is mirrored by the use of the projection π_1.

4.3 Provability in XCC

Theorem: The properties (U1)–(αh) from Section 4.2. are provable in XCC relative to the context *Co*.
Proof: All the propositions obtained by lifting the module algebra laws to type theory (instead of being assumed as axioms like in the module algebra) have been proved in the LEGO system, using the *Extended Calculus of Constructions* XCC [Luo 89]. The complete proof as LEGO-code is available from the authors. □
Once the module algebra laws are *adequately* rephrased in type theory the proofs are not very difficult. There are, however, some subtle difficulties when proving them in XCC. The next Section discusses these problems in detail.

Proof of (E2) In the following let $\pi_{x.y}(u)$ be an abbreviation for $\pi_y(\pi_x(u))$. To give an impression of how proofs look like we present a proof sketch for (E2). It is appropriate to give the motivation for using "split" versions of predicates and explain some difficulties arising from intensional type theory. If we expand the definitions of the operations then (E2) becomes

$\forall P{:}S_B.\ \forall Q{:}S_C.\ \forall s{:}A.$

$$
\begin{aligned}
&[\exists\ t{:}S_B.\ P\ \langle \pi_1(t),\ \pi_2(t)\rangle \wedge (s{=}\ \pi_1(t))] \wedge \\
&[\exists\ u{:}S_C.\ R\ \langle \pi_1(u),\ \pi_2(u)\rangle \wedge (s{=}\ \pi_1(u))] \\
\Leftrightarrow \\
&[\exists\ w{:}S_{AC}.\ (\ P\ \langle \pi_1(w),\ \pi_{1.2}(w)\rangle \wedge R\ \langle \pi_1(w),\ \pi_{2.2}(w)\rangle\ ) \wedge (s = \pi_1(w))].
\end{aligned}
$$

The "$\Leftarrow$"-direction is easy to verify. The "realizers" for the existential quantifiers of the left hand side are $t =_{def} \langle \pi_1(w), \pi_{1.2}(w)\rangle$ and $u =_{def} \langle \pi_1(w), \pi_{2.2}(w)\rangle$.
The "$\Rightarrow$"-direction is a little more complicated to prove. We suspect the realizer of the existential quantifier to be $w =_{def} \langle \pi_1(t),\ \langle \pi_2(t),\ \pi_2(u)\rangle\rangle$. But although $\pi_1(t) = s = \pi_1(u)$ holds, the type checker is not able to accept this tuple as well-typed. This is because in an intensional type theory like XCC the rule:

$$\frac{A{:}Type;\ B, C{:}A \to Type;\ x{:}S_B;\ y{:}S_C\ \vdash \pi_1(x) = \pi_1(y) \in A}{A{:}Type;\ B, C{:}A \to Type;\ x{:}S_B;\ y{:}S_C\ \vdash \langle x, \pi_2(y)\rangle \in S_{AC}}$$

is not derivable, i.e. from the fact that one can prove $\pi_1(x) = \pi_1(y)$ with Leibniz equality, it does not automatically follow that the types $C(\pi_1(x))$ and $C(\pi_1(y))$ are intensionally equal. But since we are only proving existence and need not construct the object explicitly, we can fortunately employ the following **Lemma:**

$$
\begin{aligned}
&\forall P{:}Pred(S_B).\ \forall x{:}A. \\
&\qquad (\exists y{:}(B\ x).\ \ P\langle x,\ y\rangle) \Leftrightarrow (\exists\ s{:}S_B.\ P\ \langle \pi_1(s), \pi_2(s)\rangle)\ \wedge\ (x = \pi_1(s))
\end{aligned}
$$

the proof of which has been provided by Z.Luo [Luo 92]. Now after discharging P, R and s we can apply the "$\Leftarrow$" direction of this lemma twice (instantiating P with P and x with s) and yield as new goal

$$\exists y_1{:}B(s).\ P\ \langle s,\ y_1\rangle \wedge \exists y_2{:}C(s).\ R\ \langle s,\ y_2\rangle \Rightarrow [\exists\ w{:}S_{AC}.\ldots]$$

This Lemma is used to "decompose" the objects t into two separated objects s and y_1 with dependent types. The same holds for u with s and y_2. Sinced both objects y_1 and y_2 depend on the same s one can now build the realizer of the existential quantifier on the right hand side of the implication, namely $w =_{def} \langle s, \langle y_1, y_2\rangle\rangle$ and the rest of the proof is straightforward.
Note that the splitting of P and Q is necessary because after substituting the realizer for w we get the proposition $P\ \langle \pi_1(t), \pi_2(t)\rangle$ which cannot be deduced from $P(t)$.

5 Conclusion

We have expressed the laws of module algebra formally in type theory. As in XCC one cannot derive surjective pairing for dependent sum types, the module algebra laws had to be presented in a "split" version, i.e. instead of $P(s)$ one writes $P(\langle \pi_1(s), \pi_2(s)\rangle)$. But this problem could be avoided by chosing Martin Löf's notation of dependent sum types or alternatively by introducing surjective pairing into XCC as an axiom.

As type theory can express in a quite flexible way functions and predicates on deliverables it does not seem necessary to restrict to the specific laws of module algebra. Nevertheless these can be considered as a starting point for a library of theorems useful for the composition of large systems and might be reused for implementation proofs.

Acknowledgement: Thanks to Martin Wirsing for moral support, for reading several drafts and for his comments on how to improve readability. Thanks also to Zhaohui Luo for answering detailed questions about XCC and for the enlightenment he gave us with the proof of what we have baptized *Luo's Lemma*.

References

[Bergstra et al. 89] J. A. Bergstra, J. Heering, P. Klint: *Algebraic Specification.* ACM Press, New York, 1989.

[Burstall, Goguen 80] R.M. Burstall, J.A. Goguen: *The semantics of Clear, a specification language.* In: Proc. Advanced Course on Abstract Software Specification. LNCS 86, Springer, Berlin, 1980.

[Burstall, McKinna 92] R.M. Burstall, J. McKinna: *Deliverables: a categorical approach to program development in type theory.* ECS-LFCS-92-242, Dept. of Computer Science, Edinburgh University, 1992.

[McKinna 92] J. McKinna: *Deliverables: a categorical approach to program development in type theory.* Ph.D. Thesis, University of Edinburgh, 1992. Also appeared as ECS-LFCS-92-247, Dept. of Computer Science, Edinburgh University, 1992.

[Leszczylowski, Wirsing 91] J. Leszczylowski, M. Wirsing: *Polymorphism, Parameterisation and Typing: An Algebraic Specification Perspective.* In: Proc. of the 8th STACS, LNCS 480, Springer, Berlin, p. 1 - 15.

[Luo 89] Z. Luo: *ECC, an Extended Calculus of Constructions.* In: Proc. of the Fourth Ann. Symp. on Logic in Computer Science (LICS), IEEE, 1989, p. 385 - 395.

[Luo 91] Z. Luo: *Program specification and data refinement in type theory.* In: Proc. of the Fourth Intern. Conf. on the Theory and Practice of Software Development (TAPSOFT), LNCS 493, Springer, Berlin, 1991, p. 143 - 168.

[Luo 92] Z. Luo: private communication.

[Luo et al. 92] Z. Luo, R. Pollack: *LEGO Proof Development System: Users' Manual.* ECS-LFCS-92-211, Dept. of Computer Science, Edinburgh University, 1992.

[Nordström et al. 90] B. Norström, K. Petersson, J.M. Smith: *Programming in Martin Löf's Type Theory.* Oxford University Press, Oxford, 1990.

[Streicher 88] T. Streicher: *Semantics of Type Theory – Correctness, Completeness and Independence Results.* Birkhäuser, Boston, 1991.

[Streicher, Wirsing 90] T. Streicher, M. Wirsing: *Dependent types considered necessary for algebraic specification languages.* In: Recent Trends in Data Type Specification, Proc. 7th International Workshop on Specification of Abstract Data Types. LNCS 534, Springer, Berlin, 1990, p. 323 - 340.

[Wirsing 86] M. Wirsing: *Structured algebraic specifications: a kernel language.* Theoretical Computer Science 42, 1986, p. 123 - 249.

[Wirsing 89] M. Wirsing: *Algebraic Specification* in J.V.Leeuwen (ed.): Handbook of Theoretical Computer Science, Volume B, Elsevier, Amsterdam, 1990, p. 675 - 788.