Implementation of Derived Programs (Almost) for Free*

Maura Cerioli and Elena Zucca

DISI–Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy,
e-mail: {cerioli,zucca}@disi.unige.it

Introduction

In the process of top-down software development, an implementation step can consist of two different kinds of refinement:

- "local", i.e. replacing a module A by a more specific module B which simulates the behavior of A;
- "global", i.e. passing from a more abstract specification or programming language, say $\mathcal{I}$, to a less abstract, say $\mathcal{I}'$.

A property that usually holds in practice is that these two kinds of refinement can be composed, i.e., if a module A is correctly implemented by B, then all the programs in $\mathcal{I}$ using A can be correctly transformed in programs in $\mathcal{I}'$ using B, provided that we are able to translate linguistic constructs from $\mathcal{I}$ to $\mathcal{I}'$ (in other words, we get "for free" the implementation of derived programs).

Our aim is to give a model for this situation independent from the particular formalisms $\mathcal{I}$ and $\mathcal{I}'$ which are involved, in the spirit of the theory of institutions [3]. Indeed, commonly used notions of refinement of formalisms (see [7] for further references, too) do not support this intuition, since the translation of expressions from $\mathcal{I}$ to $\mathcal{I}'$ is not given once and for all, but depends on (is parameterized by) the specific signature.

The framework we propose is partly similar to that of parchments, that are syntactic representations of institutions where expressions over a signature Σ are terms over an algebraic signature $Lang(\Sigma)$ (see e.g. [2]); but we take the stronger uniformity requirement that this algebraic signature is independent from the specific signature Σ, which only impacts on the choice of variables. Moreover, since our aim is to have a notion of formalism including both specification and programming languages, the expression we consider are not just boolean sentences, but are classified by their *types*. Putting the two things together, we get a notion of *typed uniform parchment* (Section 1), which on one side seems general enough for capturing most common institutions, on the other side allows to express notions closer to programming languages, in particular the factorization of an implementation step in a global and a local part mentioned above (Section 2).

* Partially supported by Murst 40% - Modelli della computazione e dei linguaggi di programmazione and CNR - Formalismi per la specifica e la descrizione di sistemi ad oggetti.

Finally, the notion of implementation presented here allows to relate individual values within the carriers of the involved models, generalizing the original approach by Hoare [4]. Since this *concrete data-type implementation* maps individual elements, it is possible to verify the correctness of an implementation w.r.t. a singled out function of the signature, while the available notions of implementation require to take into account whole models (or even specifications) at one time. This is, at our knowledge, the first attempt at rephrasing concrete data-type implementation in an institutional framework.

1 Typed Uniform Parchments

The institution theory has been originally developed in order to provide a flexible formalism to represent specification languages focusing on the validity relation between boolean sentences and models.

In order to represent not only axiomatic frameworks, but also a broader range of applications, like database query systems, knowledge representation systems and programming languages, institutions can be *generalized* (see e.g. [2]) replacing boolean sentences by "expressions" (we will adopt this term from now on) to be evaluated in some generic category of *values*.

Definition 1 [2]. A *generalized institution* is a tuple $(\mathbf{Sign}, \mathit{Exp}, \mathit{Mod}, [\![\_]\!])$, where:

- $\mathbf{Sign}$ is a a category,
- $\mathit{Exp}\colon \mathbf{Sign} \to \mathbf{Set}$ is a functor giving the *expressions* over a signature,
- $\mathit{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$ is a functor giving the *models* over a signature,
- $[\![\_]\!]\colon |\mathit{Mod}| \times \mathit{Exp} \stackrel{..}{\longrightarrow} \mathcal{V}$ is an extra-natural transformation, giving the *evaluation* of an expression in a model, with $\mathcal{V}$ the *universe of admissible values*; the application of $[\![\_]\!]_\Sigma$ to arguments $ex \in \mathit{Exp}(\Sigma)$ and $M \in |\mathit{Mod}(\Sigma)|$ will be denoted by $[\![ex]\!]_\Sigma^M$.

The extra-naturality of evaluation means that for each signature morphism $\sigma\colon \Sigma \to \Sigma'$, each Σ-expression ex and each Σ'-model M'

$$[\![\mathit{Exp}(\sigma)(ex)]\!]_{\Sigma'}^{M'} = [\![ex]\!]_\Sigma^{\mathit{Mod}(\sigma)(M)}.$$

In order to give a natural description of programming languages, we endow generalized institutions with a notion of *typing*. A similar notion is that of *concrete institutions* (see e.g. [1]), i.e. standard institutions with boolean sentences, where signatures have an underlying set of *sorts* or *types* and models over a signature Σ with types S have an underlying *carrier* which is an S-sorted set.

Definition 2. A *typed institution* $\mathcal{TGI}$ consists of a generalized institution $\mathcal{GI} = (\mathbf{Sign}, \mathit{Exp}, \mathit{Mod}, [\![\_]\!])$ and a *typing system* for $\mathcal{GI}$, i.e.:

- a functor $\mathit{Types}\colon \mathbf{Sign} \to \mathbf{Set}$, giving the *types* of a signature;
- a natural transformation $\tau\colon \mathit{Exp} \stackrel{.}{\longrightarrow} \mathit{Types}$, *typing* the expressions;
- an extra-natural transformation $[\_]\colon |\mathit{Mod}| \times \mathit{Types} \stackrel{..}{\longrightarrow} \mathcal{T}$, with $\mathcal{T} \subseteq \wp(\mathcal{V})$ the *universe of admissible type values*, giving the *carrier* of a type in a model; the application of $[\_]_\Sigma$ to arguments $t \in \mathit{Types}(\Sigma)$ and $M \in |\mathit{Mod}(\Sigma)|$ will be denoted by $[t]_\Sigma^M$;

satisfying, for each $\Sigma \in |\mathbf{Sign}|$, $M \in |Mod(\Sigma)|$ and $ex \in Exp(\Sigma)$, the following *type preservation* condition:

$$[\![ex]\!]^M_\Sigma \in [\tau_\Sigma(ex)]^M_\Sigma.$$

Parchments have been introduced in [2] as syntactical presentations of institutions, allowing not only to avoid the (often tedious and/or difficult) check that the satisfaction condition holds, but also to combine institutions, presented by parchments, in a more convenient way (see e.g. [5]).

The intuition behind parchments is to define the set of the sentences over a signature Σ by initiality, i.e. as the set of the terms of a distinguished sort in an algebraic signature $Lang(\Sigma)$, where $Lang$ is a functor intuitively associating a *language* with any signature. Moreover, a model over Σ can be defined by giving a morphism $M: \Sigma \to \Gamma$ where Γ is a distinguished signature, intuitively the "universal language". The validity of sentences in a model is then derived from the evaluation of the term algebra $W_{Lang(\Gamma)}$ into a fixed algebra G^V, intuitively the "semantical universe".

The *typed uniform parchments* (shorty tu parchments from now on) we present in this section differ from parchments in two respects. First of all, as they describe typed institutions, tu parchments have two levels of language: the *type* language, describing for each signature the set of acceptable types, and the *value* language, describing the (typed) expressions denoting values. The two levels also apply to the semantic part; so we have two universal languages and two algebras giving the semantical universes.

Second, we require a much stronger uniformity in the way of associating a language with a given signature. Indeed, in most commonly used institutions and in programming languages too, there is a unique language fixed once and for all, and a specific signature only contributes in providing a family of symbols which play the role of variables. This is patent in programming languages, where correct programs (expressions) using a module are usually determined by a fixed grammar (the language description) and a set of identifiers, depending on the individual module. But this also applies to specification frameworks. Let us consider, for instance, the ground terms over an algebraic signature $\Sigma = \langle S, O\rangle$. They can be seen as the terms over another signature $AV\Sigma(S)$, with sorts $\{term(s) \mid s \in S\} \cup \{op(s_1 \dots s_k \to s) \mid s_1, \dots, s_k, s \in S\}$ and an operation $\_(\_, \dots, \_): op(s_1 \dots s_k \to s) \times term(s_1) \times \dots \times term(s_k) \to term(s)$ for any $s_1, \dots, s_k, s \in S$. Each operation $f \in O_{s_1 \dots s_k, s}$ is seen as a variable of sort $op(s_1 \dots s_k \to s)$. Moreover, the sorts of the signature $AV\Sigma(S)$ are, in turn, terms over a signature $AT\Sigma$, with operations $term(\_)$ and $op(\_ \dots \_ \to \_)$, over the set S of type variables. Hence the specific signature Σ only determines the sets S and O of variables to be used respectively to build the *type* and the *value* expressions. Thus, in tu parchments the language components of both levels factorize into a functor, providing for each signature its *names*, and an algebraic signature, describing how these names are used to build the (type and value) expressions.

Notation 1: Indexed Sets. Let us denote by $SSet: \mathbf{Set}^{op} \to \mathbf{Cat}$ the *indexed set* functor, associating with each set S the category of S-sorted families of sets (and S-sorted families of functions as arrows) and with each function renaming the indexes the *reduct*. Moreover, **SSet** denotes the flattening of *SSet*, that is the category of sorted sets, where an object is a pair $\langle S, X\rangle$ with S a set and X an S-family of sets, and an arrow from $\langle S, X\rangle$ into $\langle S', X'\rangle$ is a pair $\langle f: S \to S', \{h_s: X_s \to X'_{f(s)}\}_{s\in S}\rangle$. Finally, $Sorts$: **SSet** $\to$ **Set** denotes the functor giving the sorts of a sorted set.

Here, we chose to use (many-sorted) algebraic signatures to describe the language components, for simplicity. However, the results we present are only based on the existence of the free algebra construction, and the commutativity of the reduct functor w.r.t. the carrier functor.

Notation 2: Algebraic Signatures. We denote by **AlgSign** the category of algebraic many-sorted signatures and by $Sorts$: **AlgSign** $\to$ **Set** the functor giving the sorts of an algebraic signature; when clear from the context, we will use σ for $Sorts(\sigma)$.

The functor $\mathbf{ALG}: \mathbf{AlgSign}^{op} \to \mathbf{Cat}$ associates with each $\Sigma \in |\mathbf{AlgSign}|$ the category of the many-sorted total algebras over Σ.

For each $\Sigma \in |\mathbf{AlgSign}|$ let $W_\Sigma: SSet(Sorts(\Sigma)) \to \mathbf{ALG}(\Sigma)$ denote the left adjoint of the carrier functor $|\_|: \mathbf{ALG}(\Sigma) \to SSet(Sorts(\Sigma))$, with unit η^Σ and counit ϵ^Σ.

Moreover, for each $A \in |\mathbf{ALG}(\Sigma)|$ and $f: X \to |A|$, we denote by $eval_\Sigma^{A,f}: W_\Sigma(X) \to A$ the unique free extension of f, that is $eval_\Sigma^{A,f} = \epsilon^\Sigma_A \cdot W_\Sigma(f)$. Finally, if $A = W_{\Sigma'}(Y)_{|\sigma}$ for some morphism $\sigma: \Sigma \to \Sigma'$ in **AlgSign**, then $eval_\Sigma^{A,f}$ will be denoted by $W_\sigma(f)$.

Let us collect a few properties of $W_\sigma(f)$, that will be used in the next section.

Lemma 3. *For each* $\sigma: \Sigma \to \Sigma'$, $\sigma': \Sigma' \to \Sigma''$ *in* **AlgSign**, $f: X \to |W_{\Sigma'}(Y)_{|\sigma}|$, $f': Y \to |W_{\Sigma''}(Z)_{|\sigma'}|$ *and* $f'': X \to Y_{|\sigma}$ *in* **Set**:

1. $W_{\sigma'\cdot\sigma}(|W_{\sigma'}(f')_{|\sigma}| \cdot f) = W_{\sigma'}(f')_{|\sigma} \cdot W_\sigma(f)$
2. $|W_{\sigma'}(f')_{|\sigma}| \cdot (\eta^{\Sigma'}_Y)_{|\sigma} = f'_{|\sigma}$
3. $W_{\sigma'\cdot\sigma}(f'_{|\sigma} \cdot f'') = W_{\sigma'}(f')_{|\sigma} \cdot W_\sigma((\eta^{\Sigma'}_Y)_{|\sigma} \cdot f'')$

In the following a *type language* will be a signature $T\Sigma$ with fixed sorts n, e of the *name* and *expression* types, respectively. Intuitively, the $T\Sigma$-terms of sort e represent the types of expressions, while those of sort n represent auxiliary types, whose elements will be used for constructing the expressions. In most common cases, the auxiliary types collect function and procedures, like in the following example of many-sorted signatures

Example 1.

$$AT\Sigma = \quad \textbf{sorts } n, e \quad \textbf{opns } \{op(\_ \ldots \_ \to \_): e^k \times e \to n \mid k \geq 0\}$$

Each type language implicitly defines a category of sets with *substitutions* as arrows, where type names are allowed to be expanded into type expressions. This is technically described in terms of the *Kleisli* category for the monad induced by the adjunction between the carrier functor and the free algebra construction.

Notation 3: Type Languages. We denote by **AlgSign***, the sub-category of **AlgSign** with fixed sorts n, e of the *name and expression types*, respectively, and signature morphisms preserving such fixed sorts. Signatures in **AlgSign*** will be called *type languages*.

The following notations are introduced for a given type language $T\Sigma$.

Adjunction Let us denote by $F^{T\Sigma} = W_{T\Sigma}(\_ : e)$ the left adjoint to $G^{T\Sigma} = |\_|_e$ with $\eta^{T\Sigma}_e = (\eta^{T\Sigma}_{\_:e})_e$ as unit and $\epsilon^{T\Sigma}_e = \epsilon^{T\Sigma} \cdot W_{T\Sigma}(E_{|\_|})$ as counit of the adjunction, obtained by composing the usual adjunction between indexed sets and algebras with the following adjunction $\langle \_ : e, \_{}_e, id, E\rangle$ between sets and indexed sets:

- $\_ : e$: $\mathbf{Set} \to SSet(Sorts(\Sigma))$ associates any X with the family of sets that are all empty but for the index e, for which X is yielded; analogously for functions;
- $\_{}_e$: $SSet(Sorts(\Sigma)) \to \mathbf{Set}$ is the projection over the index e;
- E is, for each family of sets X, the embedding into X of the family having all components empty but for the index e, for which X_e is yielded.

Substitution category Let us denote by $\mathbf{Set}_{T\Sigma}$ the *Kleisli*'s category over the monad induced by the adjunction between $F^{T\Sigma}$ and $G^{T\Sigma}$, having sets as objects, the functions from X into $|W_{T\Sigma}(Y : e)|_e$ as arrows from X to Y, $\eta^{T\Sigma}_e$ as identity, and composition defined by $g \overset{T\Sigma}{\circ} f = G^{T\Sigma}(\epsilon^{T\Sigma}_{e\,F^{T\Sigma}(Z)} \cdot F^{T\Sigma}(g)) \cdot f$ for each $f: X \to Y$ and $g: Y \to Z$.
Moreover, for each $f: X \to Y$ in $\mathbf{Set}$ we will denote by $f_{T\Sigma} = \eta^{T\Sigma}_{e\,Y} \cdot f: X \to Y$ its corresponding arrow in $\mathbf{Set}_{T\Sigma}$.

Name and Expression Functors The functors $|W_{T\Sigma}(\_ : e)|_n: \mathbf{Set}_{T\Sigma} \to \mathbf{Set}$ and $|W_{T\Sigma}(\_ : e)|_e: \mathbf{Set}_{T\Sigma} \to \mathbf{Set}$, give the sets of *name* and *expression* types and their homomorphical translations, that are the composition of the projections over the n and e components of the carrier with the generalization[2] of $F^{T\Sigma}$. We will also denote by $|W_{T\Sigma}(\_ : e)|_{n \uplus e}: \mathbf{Set}_{T\Sigma} \to \mathbf{Set}$ the coproduct of such functors, yielding on each set the disjoint union of the nameable and expression types, with silent injections.

The intuition behind the choice of symbols in the following definition is that a T is used to decorate the parts regarding the *type level*, an N for the *auxiliary names* and their valuations and a V for the elements of the *value level* that are directly involved in building and evaluating value expression.

Definition 4. A *typed uniform parchment*, from now on tu parchment, is a tuple $\mathcal{P} = (\mathbf{Sign}, TN, T\Sigma, \mathcal{TN}, G^T, \nu^T, N, V\Sigma, \mathcal{N}, G^V, \nu^N)$ where

Signatures Sign is a category of *signatures*,
Type level

- TN: $\mathbf{Sign} \to \mathbf{Set}$ is a functor giving the *type names* of a signature,
- $T\Sigma \in |\mathbf{AlgSign}^*|$ is the *type language*,
- $\mathcal{TN}$ is a set giving the *universe of types*;
- G^T is a model of $T\Sigma$ term-generated by the valuation $\nu^T: \mathcal{TN} \to |G^T|_e$, s.t. $|G^T|_n, |G^T|_e \subseteq |\mathbf{Set}|$;

Value level

- N: $\mathbf{Sign} \to \mathbf{SSet}$ is a functor giving the *(value) names* of a signature s.t. $Sorts \cdot N = |W_{T\Sigma}(TN(\_) : e)|_n$;
- $V\Sigma$: $\mathbf{Set}_{T\Sigma} \to \mathbf{AlgSign}$ is a functor giving the *(value) languages*, s.t. $Sorts \cdot V\Sigma = |W_{T\Sigma}(\_ : e)|_{n \uplus e}$ (*well-typedness*)
- $\mathcal{N}$ is a sorted set giving the *universe of values*, s.t. $Sorts(\mathcal{N}) = |W_{T\Sigma}(\mathcal{TN} : e)|_n$;

[2] If $\langle F, G, \eta, \epsilon\rangle: \mathbf{X} \to \mathbf{A}$ is an adjunction, then F may be generalized to a functor $F^+: \mathbf{X}_{GF} \to \mathbf{A}$ by $F^+(f: X \to Y) = \epsilon_{F(Y)} \cdot F(f)$.

– G^V is a model of $V\Sigma(\mathcal{TN})$ term-generated by the valuation $\nu^N : \mathcal{N} \to |G^V|$ s.t. for any $s \in |W_{T\Sigma}(\mathcal{TN} : e)|_{n \uplus e}$, $|G^V|_s = eval_{T\Sigma}^{G^T, \nu^T}(s)$.

Since the sort component of the indexed set morphism given by the application of N or $V\Sigma$ to arrows is fixed, in the following we will omit it, provided that no ambiguity arises.

Note that the value languages are indexed on sets instead of signatures. This, together with the well-typedness condition, guarantees a strong uniformity. In particular, if $|\mathbf{Set}|$ is the power-set of some universe $\mathcal{U}$, intuitively representing all possible *type names*, then it is possible to prove that $V\Sigma(\mathcal{U})$ and the family of the $V\Sigma(f)$ for f an endomorphism of $\mathcal{U}$ describe $V\Sigma$ up to isomorphism, so that $V\Sigma$ can be equivalently presented by *one* signature and a bunch of morphisms.

Let us illustrate the definition on the example of a simple algebraic language. Since standard many-sorted algebras have a very poor language, with only function application, we have enriched them by a few simple constructs, in order to better show the features of tu parchments. The languages defined by $AT\Sigma$ and $AV\Sigma$ do not take into account static constraints (i.e. also non well-formed expressions are obtained by the λ-abstraction construct applied to terms on a larger set of variables, which are semantically evaluated to a special $\perp$ value). Static constraints could be easily enforced adding axioms, that is using a category of specifications instead of $\mathbf{AlgSign}^*$. Here we take this approach for sake of simplicity.

Example 2. Let us describe the components of the parchment $\mathcal{ALG}$. Let us fix a universe $\mathcal{X}$ of variables to be used in value expressions.

Signatures $\mathbf{ASign} = \mathbf{AlgSign}$,
Type Level

– $ATN = Sorts$,
– $AT\Sigma$ is described in Example 1;
– $\mathcal{ATN}$ is some universe of sets; let $\mathcal{V}$ denote the universe of all the elements of sets in $\mathcal{ATN}$ and $Val = \{V \mid V : Dom(V) \to \mathcal{V}\}$ with $Dom(V) \subseteq \mathcal{X}$ denote the set of *environments*;
– $A\nu^T(A) = (Val \to A^\perp)$, where $A^\perp = A \cup \{\perp\}$, for each $A \in \mathcal{ATN}$
$$op^{AG^T}((Val \to A_1^\perp) \ldots (Val \to A_n^\perp) \to (Val \to A^\perp)) = (A_1 \times \ldots \times A_n \to A^\perp)$$

Value Level

– the functor AN on a signature $\langle S, O\rangle$ yields the indexed set having as component of index $op(s_1 \ldots s_n \to s)$ the set $O_{s_1 \ldots s_n, s}$ and is analogously defined on morphisms;
– For each $S \in |\mathbf{Set}|$,
$AV\Sigma(S) =$
sorts $|W_{AT\Sigma}(S : e)|_{n \uplus e}$
opns $\{\_(\_, \ldots, \_) : op(s_1 \ldots s_k \to s) \times s_1 \times \ldots \times s_k \to s \mid k \geq 0, s, s_1, \ldots, s_k \in S\}$
$\cup \{x : \to s \mid x \in \mathcal{X}, s \in S\}$
$\cup \{\lambda x_1 \ldots x_k . \_ : s \to op(s_1 \ldots s_k \to s) \mid k \geq 0, s, s_1, \ldots, s_k \in S\}$

The translation along some $f: S \to |\underset{AT\Sigma}{W}(S' : e)|_e$ is $|\underset{AT\Sigma}{W}(f : e)|_{n \uplus e}$ on sorts and the family of identities on operations.

- $\mathcal{AN}_{op(A_1 \dots A_n \to A)} = \{f: A_1 \times \dots \times A_n \to A^\perp\}$, for each $A, A_1, \dots, A_n \in \mathcal{ATN}$
- $A\nu^N(f) = f$, for each $f \in \mathcal{AN}$

  for each $\_(\_, \dots, \_): op(A_1 \dots A_n \to A) \times A_1 \times \dots \times A_n \to A$,

$$f(t_1, \dots, t_n)^{AG^V} = \lambda V. f^\perp(t_1(V), \dots, t_n(V))$$

  where $f: A_1^\perp \times \dots \times A_n^\perp \to A^\perp$ is defined by $f^\perp(a_1, \dots, a_n) = f(a_1, \dots, a_n)$ if $a_i \in A_i$ for $i = 1 \dots n$, $\perp$ otherwise

  $x^{AG^V} = \lambda V. V^\perp(x)$, for $V^\perp(x) = x$ if $x \in Dom(V)$, $\perp$ otherwise, $\forall x \in \mathcal{X}$

  for each $\lambda x_1 \dots x_k.\_: A \to op(A_1 \dots A_k \to A)$,

$$(\lambda x_1 \dots x_k.t)^{AG^V} = \lambda v_1 \dots v_k.t(V)$$

  with $Dom(V) = \{x_1, \dots, x_k\}$ and $V(x_i) = v_i$, $i = 1 \dots k$.

Each tu parchment presents a typed institution, where the types of each signature Σ are uniformly built as terms over the signature $T\Sigma$, using the type names S of Σ as variables. Analogously, the expressions over Σ are uniformly built as terms over the signature $V\Sigma(S)$, using the names of Σ as variables.

Proposition 5. *Each tu parchment*

$$\mathcal{P} = (\mathbf{Sign}, TN, T\Sigma, \mathcal{TN}, G^T, \nu^T, N, V\Sigma, \mathcal{N}, G^V, \nu^N)$$

defines a typed institution $(\mathbf{Sign}, Exp, Mod, [\![\_]\!], |\underset{T\Sigma}{W}(TN(\_) : e)|_e, \tau, [\_])$ *as follows.*

- Exp: $\mathbf{Sign} \to \mathbf{Set}$ *is defined by:*

  On objects: *for each* $\Sigma \in |\mathbf{Sign}|$ *with* $TN(\Sigma) = S$, $N(\Sigma) = N$,

$$Exp(\Sigma) = \{\langle ex, s\rangle \mid ex \in |\underset{V\Sigma(S)}{W}(N)|_s \text{ and } s \in |\underset{T\Sigma}{W}(S : e)|_e\}$$

  On arrows: *for each arrow* $\sigma: \Sigma \to \Sigma'$ *in* $\mathbf{Sign}$, *with* $TN(\sigma) = f: S \to S'$, $N(\sigma) = h: N \to N'_{|V\Sigma(f)}$,

$$Exp(\sigma)(ex, s) = \langle \underset{V\Sigma(f_{T\Sigma})}{W}(\eta^{V\Sigma(S')}_{N'}{}_{|V\Sigma(f_{T\Sigma})} \cdot h)(ex), |\underset{T\Sigma}{W}(f : e)|_e(s)\rangle,$$

- Mod: $\mathbf{Sign}^{op} \to \mathbf{Cat}$ *is defined by*

  On objects: *for each* $\Sigma \in |\mathbf{Sign}|$ *with* $TN(\Sigma) = S$, $N(\Sigma) = N$, $Mod(\Sigma)$ *is the discrete category whose objects are*

$$\{\langle m^T, m^N\rangle \mid m^T: S \to |\underset{T\Sigma}{W}(\mathcal{TN} : e)|_e, m^N: N \to |\underset{V\Sigma(\mathcal{TN})}{W}(\mathcal{N})_{|V\Sigma(m^T)}|\}$$

  On arrows: *for each arrow* $\sigma: \Sigma \to \Sigma'$ *in* $\mathbf{Sign}$, *with* $TN(\sigma) = f: S \to S'$, $N(\sigma) = h: N \to N'_{|V\Sigma(f)}$, *and each* $\langle m'^T, m'^N\rangle \in |Mod(\Sigma')|$

$$Mod(\sigma)(\langle m'^T, m'^N\rangle) = \langle m'^T \cdot f, m'^N_{|V\Sigma(f)} \cdot h\rangle.$$

– $[\![\_]\!]\colon |Mod| \times Exp \xrightarrow{\cdot\cdot} \mathcal{V}$, *with* $\mathcal{V} = \bigcup_{s \in |W_{T\Sigma}(\mathcal{TN}:e)|_e} |G^V|_s$, *is the extranatural transformation whose components are defined as follows:*
for each $\Sigma \in |\mathbf{Sign}|$, *with* $TN(\Sigma) = S$, $N(\Sigma) = N$, *each model* $\langle m^T, m^N\rangle$ *over* Σ *and each* $s \in |W_{T\Sigma}(S : e)|_e$

$$[\![\langle \_, s\rangle]\!]_\Sigma^{\langle m^T, m^N\rangle} = eval^{G^V, \nu^N}_{V\Sigma(\mathcal{TN})|\,|W_{T\Sigma}(m^T:e)|_e(s)} \cdot |W(m^N)_{V\Sigma(m^T)}|_s.$$

– $\tau\colon Exp \xrightarrow{\cdot} |W_{T\Sigma}(TN(\_) : e)|_e$ *is the natural transformation whose components are the projections on the second component;*
– $[\_]\colon |Mod| \times |W_{T\Sigma}(TN(\_) : e)|_e \xrightarrow{\cdot\cdot} \mathcal{T}$, *with* $\mathcal{T} = |G^T|_e$ *is the extranatural transformation defined, for each signature* Σ *and* Σ*-model* $\langle m^T, m^N\rangle$, *by:*

$$[\_]_\Sigma^{\langle m^T, m^N\rangle} = eval^{G^T, \nu^{T^{T\Sigma}} \circ m^T}_{T\Sigma}.$$

In the following we will use *models of a tu parchment* over a signature Σ for the models over Σ in the typed institution presented by the tu parchment.

Example 3. Let us sketch another example of tu parchment, $\mathcal{FUN}$, presenting a toy functional programming language `FUN`, whose modules have the form

$tid_1 = te_1, \ldots, tid_n = te_n$
$f_1(x^1_1 : te^1_1; \ldots; x^1_{n_1} : te^1_{n_1}) : te'_1 = exp_1$
$\ldots$
$f_m(x^m_1 : te^m_1; \ldots; x^m_{n_m} : te^m_{n_m}) : te'_m = exp_m$

where tid (possibly decorated, as for other metavariables) ranges over type identifiers, f over function identifiers, te over type expressions defined by:

$$te ::= \quad tid \mid \texttt{int} \mid \langle l_1 : te_1, \ldots, l_n : te_n\rangle \mid \texttt{array}\ [0..N]\ \texttt{of}\ \ te,$$

(with N any positive integer constant and l_i field identifiers), x over variables, exp over expressions defined by the grammar

$$\begin{array}{rl} exp ::= & x \mid \texttt{zero} \mid \texttt{succ}(exp) \mid \texttt{pred}(exp) \mid \langle l_1 : exp_1, \ldots, l_n : exp_n\rangle \mid [\,] \mid \\ & exp_1[exp_2/exp_3] \mid exp.l \mid exp_1[exp_2] \mid f(exp_1, \ldots, exp_n) \mid \\ & \texttt{if}\ \ exp_1 = exp_2\ \ \texttt{then}\ \ exp_3\ \ \texttt{else}\ \ exp_4. \end{array}$$

Then, signatures in **FSign** are interfaces of `FUN` modules, i.e. pairs of the form $\langle t_1 \ldots t_n, f_1(te^1_1; \ldots; te^1_{n_1}) : te'_1, \ldots, f_m(te^m_1; \ldots; te^m_{n_m}) : te'_m\rangle$; FTN and FN gives the sets of elements in the first and second component, respectively.

The type language $FT\Sigma$ has sorts n, e and one operation of sort e for each constructor of type expressions but for the first production, corresponding to the type names provided by the module. Thus,

$\texttt{int}\colon \to e$
$\langle l_1 : \_, \ldots, l_k : \_\rangle\colon e^k \to e$
$\{\texttt{array}\ [0..N]\ \texttt{of}\ \_\colon e \to e \mid N \geq 0\}$

moreover, there is a family of operations of sort n (one for each $k \geq 0$)

$\texttt{fun}(\_ \ldots \_, \_)\colon e^k \times e \to n.$

For each $S \in |\mathbf{Set}|$, the signature $FV\Sigma(S)$ has sorts $|W(\underset{FT\Sigma}{S} : e)|_{n \uplus e}$ and a family of overloaded operations (one for each possible choice of types) for each constructor of expressions, for instance, denoting $|W(\underset{FT\Sigma}{S} : e)|_e$ by TE:

$\{x\colon \rightarrow te \mid te \in TE\}$
$\{\mathtt{zero}\colon \rightarrow \mathtt{int}\}$
$\{\langle l_1 : \_, \ldots, l_n : \_\rangle\colon te_1 \times \ldots \times te_n \rightarrow \langle l_1 : te_1, \ldots, l_n : te_n\rangle \mid te_1, \ldots, te_n \in TE\}$
$\{\_(\_, \ldots, \_)\colon \mathtt{fun}(te_1 \ldots te_n, te) \times te_1 \times \ldots \times te_n \rightarrow te \mid te_1, \ldots, te_n \in TE\}$

and a family of overloaded operations corresponding to function definitions

$\{\mathtt{fun}(x_1 : te_1; \ldots; x_n : te_n) : te = \_\colon te \rightarrow \mathtt{fun}(te_1 \ldots te_n, te) \mid te, te_1, \ldots, te_n \in TE\}$

The universes $\mathcal{FTN}$ and $\mathcal{FN}$ are empty, i.e. the universal languages of the two levels consist of the ground terms over $FT\Sigma$ and $FV\Sigma(\emptyset)$ and, accordingly, the valuations $F\nu^T$ and $F\nu^N$ are the empty maps. Indeed, in this case a model is a programming module; hence it associates with the (type or value) names specified in the interface some (type or value) expressions. The algebras FG^T and FG^V correspond to the (straightforward) denotational semantics of the language; we omit them for sake of brevity.

2 Implementation and Derivation in TU Parchments

In this section, we express in the framework of tu parchments two relations between specification or programming modules playing an important role in the practice: *derivation* (a module $\overline{m}$ is defined using the primitives provided by another module m in the same formalism) and *implementation* (a module m is replaced by a "less abstract" module m', possibly changing the formalism). Moreover, we prove that implementation can be propagated "for free" to derived models, i.e. if m' implements m, and $\overline{m}$ is derived from m, then it is possible to construct an implementation $\overline{m}'$ for $\overline{m}$, provided that we are able to translate the linguistic constructs of the two formalisms.

Notation 4: Abbreviations. Let us fix for the rest of the section tu parchments

$$\mathcal{P} = (\mathbf{Sign}, TN, T\Sigma, \mathcal{TN}, G^T, \nu^T, N, V\Sigma, \mathcal{N}, G^V, \nu^N)$$
$$\mathcal{P}' = (\mathbf{Sign'}, TN', T\Sigma', \mathcal{TN}', G'^T, V'^T, N', V\Sigma', \mathcal{N}', G'^V, V'^N);$$

signatures Σ and $\overline{\Sigma}$ in $\mathcal{P}$ and Σ' in $\mathcal{P}'$, with $TN(\Sigma) = S$, $TN(\overline{\Sigma}) = \overline{S}$, $TN'(\Sigma') = S'$, $N(\Sigma) = N$, $N(\overline{\Sigma}) = \overline{N}$ and $N'(\Sigma') = N'$. Moreover, let $m = \langle m^T, m^N\rangle$, $\overline{m} = \langle \overline{m}^T, \overline{m}^N\rangle$, and $m' = \langle m'^T, m'^N\rangle$ be models over Σ, $\overline{\Sigma}$ and Σ', respectively. Finally, the composition of f and g in $\mathbf{Set}_{T\Sigma}$ ($\mathbf{Set}_{T\Sigma'}$) will be denoted by $f \circ g$ ($f \star g$) and the unit $\eta_e^{T\Sigma}$ and counit $\epsilon_e^{T\Sigma}$ ($\eta_e^{T\Sigma'}$, $\epsilon_e^{T\Sigma'}$) by η and ϵ (η', ϵ').

2.1 Derived Models

The derivation relation concerns modules within the same formalism, say m and $\overline{m}$, and holds whenever all the (type and value) names of $\overline{m}$ are defined, via a pair of maps $\chi = \langle \chi^T, \chi^N\rangle$, by expressions built over the (type and value) names of m. In this case, it is clear that the semantics of (the model corresponding to) $\overline{m}$ can be obtained by evaluating such expressions through m.

Our notion of derivation roughly corresponds to what is called implementation by *constructors* in the literature (see e.g. [7]), characterized by the fact that sorts and operations in a specification $\overline{SP}$ are defined in terms of sorts and operations of another specification SP by means of an enrichment $SP+$ of SP together with a signature morphism from $\overline{SP}$ to $SP+$. Anyway, in our approach it is not necessary to have a signature morphism, and the two steps are incorporated at a more concrete level by the mapping χ from names to expressions.

This situation happens in programming languages whenever the module $\overline{m}$ *uses* m; anyway it makes sense also in common institutions. For instance, considering many-sorted algebras, given mappings $\chi^T : \overline{S} \to S$ and $\chi^N : \overline{O} \to O$ which associate with each operation in $\overline{O}$ an open term over Σ, we can derive from an algebra A over $\Sigma = \langle S, O \rangle$ a new algebra $\overline{A}$ over $\overline{\Sigma} = \langle \overline{S}, \overline{O} \rangle$ in such a way that sort renaming χ^T is preserved. Note that in this case (as in most algebraic examples), since the type language is trivial, derivation at the level of types is just sort renaming; on the contrary, in programming languages, type names in $\overline{m}$ can be mapped in type expressions constructed over type names of m.

Definition 6. The model $\overline{m}$ is *derived* from m via

$$\chi = \begin{cases} \chi^T : \overline{S} \to |\underset{T\Sigma}{W}(S : e)|_e \\ \chi^N : \overline{N} \to |\underset{V\Sigma(S)}{W}(N)_{|V\Sigma(\chi^T)}| \end{cases} \quad \text{iff} \quad \begin{array}{l} \overline{m}^T = m^T \circ \chi^T \\ \overline{m}^N = |\underset{V\Sigma(m^T)}{W}(m^N)_{|V\Sigma(\chi^T)}| \cdot \chi^N \end{array}$$

Example 4. Let us consider the parchment $\mathcal{ALG}$ introduced in Example 2 and see a simple example of a derivation. We first define a model for the standard specification of the *stack* data-type, m_S.

Stacks of natural numbers (of maximal length K) are represented by a model $m_S = \langle m_S^T, m_S^N \rangle$ over Σ_S (having sorts *stack* and *elem* and the usual operations *empty*, *push*, *pop*, *top*); m_S^T associates $\mathbb{N} \cup \{errelem\}$ with *elem*, $\mathbb{N}^K \cup \{errstack\}$ with *stack*; m_S^N associates with operations their standard interpretations (*errstack* is used as result of *pop* on an empty stack and *push* on a full stack and *errelem* as result of *top* on an empty stack).

Then, we can derive from it a model for the richer signature Σ_S, that is the enrichment of Σ_S by *swap*: *stack* $\to$ *stack*, swapping the topmost two elements, if any, via the renaming χ that is the identity on all symbols of Σ_S but for *swap* on which it yields $\lambda s.push(top(pop(s)), push(top(s), pop(pop(s))))$.

2.2 Implementation

Having a uniform way of building types and typed expressions allows to distinguish within an implementation step two parts: a global part, translating the type and value languages of the source framework into the corresponding components of the target, and a local part, dealing with the details of the particular models and names. This is very important, because in that way the global part can be reused. In order to be able to define the global part, we need a functor relating the corresponding substitution categories.

Lemma 7. *Each arrow* $\phi^T : T\Sigma \to T\Sigma'$ *in* $\mathbf{AlgSign}^*$ *induces a functor* $F_{\phi^T} : \mathbf{Set}_{T\Sigma} \to \mathbf{Set}_{T\Sigma'}$*, defined by:*

- $F_{\phi^T}(X) = X$ *for all* $X \in |\mathbf{Set}_{T\Sigma}| = |\mathbf{Set}| = |\mathbf{Set}_{T\Sigma'}|$;
- $F_{\phi^T}(f) = G^{T\Sigma}(\epsilon_{{F^{T\Sigma'}(Y)}_{|\phi^T}} \cdot F^{T\Sigma}(\eta'_Y)) \cdot f$ *for all* $f \in \mathbf{Set}_{T\Sigma}(X, Y)$.

Moreover, if $f = {\eta_e^{T\Sigma}}_Y \cdot f'$ *for some* $f : X \to Y$ *in* $\mathbf{Set}$*, then* $F_{\phi^T}(f) = {\eta_e^{T\Sigma'}}_Y \cdot f'$.

Definition 8. A *translator* $\phi = \langle \phi^T, \phi^N \rangle$ of $\mathcal{P}$ into $\mathcal{P}'$ consists of an arrow $\phi^T : T\Sigma \to T\Sigma'$ in $\mathbf{AlgSign}^*$, and a natural transformation $\phi^N : V\Sigma \dot{\longrightarrow} V\Sigma' \cdot F_{\phi^T}$ s.t. $\mathit{Sorts}(\phi^N_S) = |W_{\phi^T}((\eta_e^{T\Sigma'})_S : e)|_{n \uplus e}$.

For each translator $\phi = \langle \phi^T, \phi^N \rangle$ and each arrow $f : S \to S'$ let us denote by ϕ_f the morphism $V\Sigma'(f_{T\Sigma'}) \cdot \phi^N_S$ translating the *higher level* signature $V\Sigma(S)$ into the *lower level* signature $V\Sigma'(S')$.

Example 5. We define a translator $\tilde{\phi}$ of $\mathcal{ALG}$ into $\mathcal{FUN}$ as

$$\langle \tilde{\phi}^T : AT\Sigma \to FT\Sigma, \tilde{\phi}^N : AV\Sigma \dot{\longrightarrow} FV\Sigma \cdot F_{\tilde{\phi}^T} \rangle,$$

where $\tilde{\phi}^T$ gives the translation from the high-level to the low-level type language, and is the identity on sorts, whose translation is fixed, and maps
$op(\_ \ldots \_ \to \_) : e^k \times e \to n$ to $\texttt{fun}(\_ \ldots \_, \_) : e^k \times e \to n$

Moreover, for each set S, $\tilde{\phi}^N_S$ gives the translation from the high-level to the low-level language of (value) expressions with types constructed starting from the type names S, and is analogously defined.

Definition 9. The model m' is an *implementation* of m w.r.t. the translator $\phi = \langle \phi^T, \phi^N \rangle$ of $\mathcal{P}$ into $\mathcal{P}'$, via ψ and abs, with

- $\psi = \begin{cases} \psi^T : S \to S' \\ \psi^N : N \to N'_{|\phi_\psi} \text{(where } \phi_\psi \text{ is a short notation for } \phi_{\psi^T}) \end{cases}$
- $abs = \{abs_s\}_{s \in |W_{T\Sigma}(S:e)|_e}$ where each $abs_s : D_s \to |W_{V\Sigma(\mathcal{TN})}(\mathcal{N})_{|V\Sigma(m^T)}|_s$, for some $D_s \subseteq |W_{V\Sigma'(\mathcal{TN}')}(\mathcal{N}')_{|V\Sigma'(m'^T) \cdot \phi_\psi}|_s$ with embedding ι_s, satisfying the following *semantic coherency* property:

  for all $t_1, t_2 \in |W_{V\Sigma'(\mathcal{TN}')}(\mathcal{N}')_{|V\Sigma'(m'^T) \cdot \phi_\psi}|_s$ if $\mathit{eval}^{G'^V, V'^N}_{V\Sigma'(\mathcal{TN}')}(t_1) = \mathit{eval}^{G'^V, V'^N}_{V\Sigma'(\mathcal{TN}')}(t_2)$, then both ($t_1 \in D_s$ iff $t_2 \in D_s$) and if $t_1, t_2 \in D_s$, then $abs_s(t_1) = abs_s(t_2)$;

iff the diagram in Fig. 1 commutes, where $\uplus$ is the coproduct in $\mathit{SSet}(|W_{T\Sigma}(S : e)|_e)$.

Example 6. We show how to express a standard implementation example, i.e. stacks (formalized by the model m_S in Example 4) by pairs array and length (formalized by a model m_R in `FUN`). Note that in our framework the high-level (implemented) and the low-level (implementing) data-types are allowed to stay "in two different worlds"; indeed, the first is expressed by a many-sorted algebra and the second by a functional program.

The model m_R corresponds to the following module

$$
\begin{tikzcd}[column sep=small]
 & \boxed{|G^{V}_{|V\Sigma(m^T)}|} & \\
 & \boxed{|W_{V\Sigma(\mathcal{TN})}(\mathcal{N})_{|V\Sigma(m^T)}|} \arrow[u, "(eval^{G^V,\nu^N}_{V\Sigma(\mathcal{TN})})_{|V\Sigma(m^T)}"] & \\
\boxed{|W_{V\Sigma(S)}(D \uplus N)|} \arrow[ur, "|W_{V\Sigma(m^T)}(abs \uplus m^N)|"] \arrow[rr, "|W_{V\Sigma'(m'^T)\cdot\phi_\psi}(\iota \uplus (m'^N_{|\phi_\psi} \cdot \psi^N))|"'] & & \boxed{|(W_{V\Sigma'(\mathcal{TN}')}(\mathcal{N}'))_{|V\Sigma'(m'^T)\cdot\phi_\psi}|} \arrow[ul, "abs"']
\end{tikzcd}
$$

Fig. 1. Implementation Diagram

```
myStack = ⟨elems : array [0..K] of int, length : int⟩, myElem = int
myEmpty : myStack = ⟨elems : [ ], length : zero⟩
myPush(s : myStack, n : myElem) : myStack = if s.length = pred(zero) then s else
    if s.length = K then ⟨elems : [ ], length : pred(zero)⟩ else
        ⟨elems : s.elems[n/s.length], length : succ(s.length)⟩
myPop(s : myStack) : myStack = if s.length = pred(zero) then s else
    if s.length = zero then ⟨elems : [ ], length : pred(zero)⟩ else
        ⟨elems : s.elems, length : pred(s.length)⟩
myTop(s : myStack) : myElem = if s.length = pred(zero) then pred(zero) else
    if s.length = zero then pred(zero) else
        s.elems[pred(s.length)]
```

We define now $\tilde{\psi}$ and $\widetilde{abs}$ s.t. m_R is an implementation of m_S (from Example 4) w.r.t. the translator $\tilde{\phi}$ (from Example 5) via $\tilde{\psi}$ and $\widetilde{abs}$.

The component $\tilde{\psi} = \langle \tilde{\psi}^T, \tilde{\psi}^N \rangle$ gives the correspondence from names in the high-level model to names in the low-level model. In this case $\tilde{\psi}^T$ maps the sorts *stack*, *elem* into the type identifiers `myStack`, `myElem`, respectively, and the operations in Σ_S into the corresponding function identifiers in the module.

The component $\widetilde{abs}$ is the so-called *abstraction map*. It is a family of functions indexed over all the expression types which one can build starting from the type names *stack*, *elem*. In this case $|W_{T\Sigma}(\{stack, elem\} : e)|_e = \{stack, elem\}$; hence $\widetilde{abs}$ has only the following two components (denoting by s' the type $\langle \texttt{elems} : \texttt{array}\ [0..K]\ \texttt{of int}, \texttt{length} : \texttt{int} \rangle$):

$$\widetilde{abs}_{elem} \colon \widetilde{D}_{elem} \subseteq |W_{FV\Sigma(\emptyset)}|_{\texttt{int}} \to |W_{AV\Sigma(\mathcal{ATN})}(\mathcal{AN})|_{\mathbb{N}}$$

$$\widetilde{abs}_{stack} \colon \widetilde{D}_{stack} \subseteq |W_{FV\Sigma(\emptyset)}|_{s'} \to |W_{AV\Sigma(\mathcal{ATN})}(\mathcal{AN})|_{(\mathbb{N}^k \cup \{errstack\})}$$

which are defined as follows

– for each $exp \in |W_{FV\,\Sigma(\emptyset)}|_{\texttt{int}}$, $exp \in \widetilde{D}_{elem}$ iff $eval^{FG^V}_{FV\,\Sigma(\emptyset)}(exp) = \lambda\rho.z$ for some $z \in \mathbb{N}$; in this case, $\widetilde{abs}_{elem}(exp) = z$
– for each $exp \in |W_{FV\,\Sigma(\emptyset)}|_{s'}$, $exp \in \widetilde{D}_{stack}$ iff $eval^{FG^V}_{FV\,\Sigma(\emptyset)}(exp) = \lambda\rho.\langle \texttt{elems} : a, \texttt{length} : l\rangle$ and $l \in \{-1 \ldots K\}$; in this case, $\widetilde{abs}_{stack}(exp) = errstack$ if $l = -1$, $a(l-1) \cdot \ldots \cdot a(0)$ otherwise.

Note that the semantic coherency property is guaranteed by the definition, that is based on the low-level semantics.

The intuitive interpretation of the implementation diagram in this particular case is the following. Take (left-bottom corner) a term in the high-level language constructed using as variables the names in the high-level model m_S and the elements of $\widetilde{D}$ (i.e. the terms in the low-level universal language which are used for the implementation). An example of such a term is $w = push(\langle \texttt{elems} : [\,], \texttt{length} : \texttt{zero}\rangle, \texttt{zero})$. Then, we obtain terms in the high-level universal language having the same semantics in the following two ways:

– first mapping the names in m_S to the corresponding names in m_R via $\tilde{\psi}$; for instance on w we get $\texttt{myPush}(\langle \texttt{elems} : [\,], \texttt{length} : \texttt{zero}\rangle, \texttt{zero})$. Then, mapping these names to their associated representation in the low-level universal language, corresponding to an expansion of each function call to its body, getting, for instance, on our example

  $\texttt{myPush}(s : \texttt{myStack}, n : \texttt{myElem}) : \texttt{myStack} =$
  $\quad \texttt{if}\ s.\texttt{length} = \texttt{pred(zero)}\ \texttt{then}\ s\ \texttt{else}$
  $\quad\quad \texttt{if}\ s.\texttt{length} = K\ \texttt{then}\ \langle \texttt{elems} : [\,], \texttt{length} : \texttt{pred(zero)}\rangle\ \texttt{else}$
  $\quad\quad\quad \langle \texttt{elems} : s.\texttt{elems}[n/s.\texttt{length}], \texttt{length} : \texttt{succ}(s.\texttt{length})\rangle$
  $\quad\quad\quad\quad (\langle \texttt{elems} : [\,], \texttt{length} : \texttt{zero}\rangle, \texttt{zero})$

  and finally abstracting the result;
– using the abstraction map to translate the low-level terms and m_S to interpret high-level names into the high-level universal language.

2.3 Deriving Implementations

We prove now that, given an implementation m' of m w.r.t. ϕ via ψ and abs, it is possible to construct an implementation $\overline{m}'$ for any model $\overline{m}$ derived from m. Referring to our working example, that means that, having implemented stacks (m_S) by records (m_R) w.r.t. $\tilde{\phi}$ via $\tilde{\psi}$ and $\widetilde{abs}$, we get "for free" an implementation also for any derived operation we can express in terms of the stack primitives, e.g. *swap* mentioned in Example 4.

This is possible under the assumption that, given a signature $\overline{\Sigma}$ in the high-level formalism (e.g. in $\mathcal{ALG}$), a signature $\overline{\Sigma}'$ in the low-level formalism (e.g. in $\mathcal{FUN}$) can be found with the same names of $\overline{\Sigma}$, up to isomorphism: in the example, a module interface in $\mathcal{FUN}$ where type and function identifiers are exactly (module some coding) sorts and operations of $\overline{\Sigma}$. Note that if the translation of the linguistic constructs building the types is not injective, then it is possible that several high-level terms $t_1, \ldots, t_k$ of sort n are translated into one low-level term t of sort n; in that case the names provided by $\overline{\Sigma}'$ of type t have to represent the names provided by $\overline{\Sigma}$ of each type t_i. Thus, the names provided by $\overline{\Sigma}'$ of type t are the disjoint union of the names provided by $\overline{\Sigma}$ of all the t_i's.

Theorem 10. *Let us assume that*

- $\overline{m}$ *is derived from* m *via* $\chi = \langle \chi^T, \chi^N \rangle$
- m' *is an implementation of* m *w.r.t. the translator* $\phi = \langle \phi^T, \phi^N \rangle$ *of* $\mathcal{P}$ *into* $\mathcal{P}'$, *via* $\psi = \langle \psi^T, \psi^N \rangle$ *and abs.*

If there exists a signature $\overline{\Sigma}'$ *in* $\mathcal{P}'$, *with* $TN'(\overline{\Sigma}') = \overline{S}'$ *and* $N'(\overline{\Sigma}') = \overline{N}'$ *s.t.*

1. *there exists an isomorphism* $\gamma\colon \overline{S} \to \overline{S}'$ *in* **Set**
2. *there exists an isomorphism* $\delta\colon \uplus\overline{N} \to \overline{N}'$, *where, for each sort* $s' \in |\underset{T\Sigma'}{W}(\overline{S}' : e)|_n$, *the set* $(\uplus\overline{N})_{s'}$ *is the coproduct of* $\{\overline{N}_s \mid \phi_\gamma(s) = s'\}$; *in the following we will denote by* $\uplus f$ *the family of functions with each* s' *component defined as the* pairing *of (that is the unique arrow whose compositions with the injections yield)* $\{f_s \mid \phi_\gamma(s) = s'\}$ *for each family of functions* $f_s\colon \overline{N}_s \to X_{s'}$

then the following properties hold

1. $\overline{m}' = \langle \overline{m}'^T, \overline{m}'^N \rangle$ *is a model of* $\overline{\Sigma}'$ *in* $\mathcal{P}'$ *where* $\overline{m}'^T = m'^T \star \chi'^T$, $\overline{m}'^N = |\underset{V\Sigma'(m'^T)}{W}(m'^N)_{|V\Sigma'(\chi'^T)}| \cdot \chi'^N$ *and*
$$\chi' = \begin{cases} \chi'^T &= \psi^T_{T\Sigma'} \star F_{\phi^T}(\chi^T) \star \gamma^{-1}_{T\Sigma'} \\ \chi'^N &= \uplus(|\underset{\phi_\psi}{W}(\psi^N)_{|V\Sigma(\chi^T)}| \cdot \chi^N) \cdot \delta^{-1} \end{cases}$$
2. $\overline{m}'$ *is derived from* m' *via* χ'
3. $\overline{m}'$ *is an implementation of* $\overline{m}$ *w.r.t.* ϕ^T *and* ϕ^N, *via* $\overline{\psi}$ *and* $\overline{abs} = abs_{|V\Sigma(\chi^T)}$, *with definition domain* $\overline{D} = D_{|V\Sigma(\chi^T)}$ *and embedding* $\overline{\iota} = \iota_{|V\Sigma(\chi^T)}$, *where*
$$\overline{\psi} = \begin{cases} \overline{\psi}^T &= \gamma \\ \overline{\psi}^N &= \delta_{|\phi_\gamma} \cdot inj \end{cases}$$
and inj is an indexed set morphism, whose sort component is $Sorts(\phi_\gamma)$ *and each* $inj_s\colon \overline{N}_s \to (\uplus\overline{N})_{\phi_\gamma(s)}$ *is the injection (in the coproduct).*

Conclusions and Further Work We have proposed a new metaframework for representing and relating formalisms, in the spirit of the theory of institutions. The overall aim is to provide a framework general enough for including common specification formalisms, but at a more concrete level, in such a way that also non-axiomatic languages are covered. In particular, within our framework it is possible to express two notions closer to programming languages, i.e. *derivation* and *implementation* and to show that implementation can be canonically extended to derived modules.

Moreover, the notion of implementation presented here is a generalization of the concrete data-type implementation introduced by Hoare [4] and allows to relate individual values. Having such relationship, it should be possible to formalize *external calls*, corresponding to models where the implementation of some name is not given, but it is imported from another model in a different language, and *program annotations*, corresponding to using the high-level logic to state properties on the elements of models in the low-level framework, whose semantics is given through the abstraction map.

Other promising applications of tu parchments are the development of fragments of typed equational first-order logic, based on the notion of typed expression evaluation providing an obvious semantics for equality, and the combinations of typed institutions through their syntactic representations, much in the spirit of [6].

Finally we plan to generalize the present approach in a forthcoming extended version allowing languages to be represented by axiomatic specifications instead of plain signatures, in order to capture static semantics constraints. Since the results presented here are based only on the existence of free objects, any choice of specifications preserving such property will carry on the results.

A somehow extended version of this paper, including proofs, is reachable from the web pages of the authors[3].

Acknowledgments. We warmly thank the anonymous referee for his/her careful reading and helpful suggestions.

References

1. M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. In H. Kirchner, editor, *CAAP '96 - 20th Coll. on Trees in Algebra and Computing*, number 1059 in LNCS, pages 241–256, Berlin, 1996. Springer Verlag.
2. J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In D. Pitt et al., editor, *Category Theory and Computer Programming*, number 240 in LNCS, pages 313–333, Berlin, 1985. Springer Verlag.
3. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for computer science. *Journ. ACM*, 39:95–146, 1992.
4. C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
5. T. Mossakowski. Using limits of parchments to systematically construct institutions of partial algebras. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *11th WADT*, number 1130 in LNCS, pages 379–393, Berlin, 1996. Springer Verlag.
6. T. Mossakowski, A. Tarlecki, and W. Pawłowski. Combining and representing logical systems. In *Category Theory and Computer Science '97*, number 1290 in LNCS, pages 177–196, Berlin, 1997. Springer Verlag.
7. F. Orejas, M. Navarro, and A. Sánchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *8th WADT*, number 655 in LNCS, pages 93–125. Springer Verlag, Berlin, 1993.

This article was processed using the LaTeX macro package with LLNCS style

[3] `http://www.disi.unige.it/person/{CerioliM,ZuccaE}/`