Extended ML: an institution-independent framework for formal program development

Donald Sannella[1] and Andrzej Tarlecki[2]

Abstract

The Extended ML specification language provides a framework for the formal stepwise development of modular programs in the Standard ML programming language from specifications. The object of this paper is to equip Extended ML with a semantics which is completely independent of the logical system used to write specifications, building on Goguen and Burstall's work on the notion of an *institution* as a formalisation of the concept of a logical system. One advantage of this is that it permits freedom in the choice of the logic used in writing specifications; an intriguing side-effect is that it enables Extended ML to be used to develop programs in languages other than Standard ML since we view programs as simply Extended ML specifications which happen to include only "executable" axioms. The semantics of Extended ML is defined in terms of the primitive specification-building operations of the ASL kernel specification language which itself has an institution-independent semantics.

It is not possible to give a semantics for Extended ML in an institutional framework without extending the notion of an institution; the new notion of an *institution with syntax* is introduced to provide an adequate foundation for this enterprise. An institution with syntax is an institution with three additions: the category of signatures is assumed to form a concrete category; an additional functor is provided which gives concrete syntactic representations of sentences; and a natural transformation associates these concrete objects with the "abstract" sentences they represent. We use the first addition to "lift" certain necessary set-theoretic constructions to the category of signatures, and the other two additions to deal with the low-level semantics of axioms.

1 Introduction

Beginning with [GTW 76], [Gut 75] and [Zil 74], work on the algebraic approach to program specification has focused on developing techniques of specifying programs (abstract data types in particular) and on formalising the notion of refinement as used in stepwise refinement (see e.g. [Ehr 79] and [EKMP 82]). The ultimate goal of this work is to provide a formal basis for program development which would support a methodology for the systematic evolution of programs from specifications by means of verified refinement steps. But so far comparatively little work has been done on applying these theoretical results to programming, with a few exceptions such as CIP-L [Bau 81], IOTA [NY 83] and Anna [LHKO 84].

The Extended ML language [ST 85a] was the result of our attempt to apply ideas about algebraic specifications in the context of the Standard ML programming language [Mil 85]. Extended ML is an

[1]Department of Artificial Intelligence, University of Edinburgh and Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh

[2]Institute of Computer Science, Polish Academy of Sciences, Warsaw

extension to Standard ML whereby axioms are permitted in module interface declarations and in place of code in module definitions. Some Extended ML specifications are executable, since Standard ML function definitions are just axioms of a certain special form. Hence Extended ML is a *wide spectrum language* in the sense of CIP-L, i.e. it can be used to express every stage in the development of a Standard ML program from the initial high-level specification to the final program itself.

The semantics of Extended ML, as sketched in [ST 85a], is expressed in terms of the primitive specification-building operations of the ASL kernel specification language [SW 83], [Wir 83]. From this semantic basis Extended ML inherits its formal notion of refinement as well as proof rules which enable refinement steps to be proved correct; moreover, the results concerning observational and behavioural abstraction in [ST 85b] can be used to obtain a better understanding of the relation between an Extended ML module and its interface.

In order to remain in the comfortable and standard framework of many-sorted algebras, it was necessary in [ST 85a] to restrict attention to the development of Standard ML programs which fit into this formalism. This meant restricting to the applicative subset of Standard ML (without assignment and exceptions) and disallowing use of polymorphic types, higher-order functions, and partial functions. Axioms were expressed in first-order predicate calculus with equality. It was hinted that these restrictions could be avoided by extending the underlying logical system appropriately; for example, partial functions could be allowed by changing the notion of an algebra (so that functions associated with operation names are not required to be total) and extending axioms and the definition of the satisfaction of an axiom by an algebra so as to provide some way of specifying the domains of functions (see [BW 82]).

In fact, there is no need to choose a particular fixed logical system. It is possible to parameterise Extended ML by an arbitrary logical system, or *institution* [GB 84]. An institution comprises definitions of signature, model (algebra), sentence (axiom) and a satisfaction relation subject to a few consistency conditions. By avoiding the choice of a particular institution when defining Extended ML, we leave open the possibility of adopting either a simple institution for use in developing a restricted class of Standard ML programs, or a more elaborate and expressive institution for use in developing programs in full Standard ML. Moreover, since the use of Extended ML as a vehicle for program development depends on viewing programs as axioms which happen to be executable, by basing Extended ML on an arbitrary institution we make it possible to develop programs in different languages within the Extended ML framework. With an appropriate change to the underlying institution, Extended "ML" becomes Extended Prolog (i.e. Prolog + modules + specifications), Extended Pascal, etc. Of course, the choice of institution determines not only the target programming language but also the rest of the logical system (including the form of non-executable axioms) which may be used to write specifications during earlier stages of program development.

Following [ST 85a], an institution-independent semantics for Extended ML may be given by defining a translation of each Extended ML specification into an ASL specification. This specification will in turn have a well-defined class of models. ASL provides a suitable basis for this enterprise since it already has an institution-independent semantics [ST 85c]. The translation from Extended ML to ASL must itself be institution-independent if the result is to express the meaning of Extended ML specifications in an arbitrary institution. This is not easy to accomplish since many of the manipulations involved in the standard case (see [ST 85a]) use set-theoretic operations like union and intersection to build signatures. Such manipulations are not possible in the context of an arbitrary institution since we have no information about the set-theoretic properties of signatures — we know only that signatures and their morphisms form a category.

In this paper we study extensions to the notion of institution which are sufficient to define an institution-independent semantics of Extended ML by translation into ASL. The basic idea will be to add the extra assumption that the category of signatures forms a *concrete category* [MacL 71], i.e. that it comes equipped with a *faithful* functor to the category of sets. This functor gives the vocabulary of names provided by each signature and the translation of names induced by each signature morphism. This enables us to view signatures as sets of names with structure which differs from one institution to another. In order to "lift" the constructions outlined in [ST 85a] to an arbitrary institution it is necessary to make a few assumptions about the properties of this functor.

Another aspect of the usual notion of institutions is inconsistent with our goals. Namely, the sentences in an institution are merely abstract objects. A semantics which covers the "low-level" details of Extended ML needs more information about sentences than this; an axiom which appears in an Extended ML specification is not an abstract entity but a syntactic representation of one. In order to take this into account we will assume that an institution comes equipped with an additional functor giving the set of syntactic representations of sentences over a vocabulary of names as well as a (partial) function which associates syntactic representations with sentences.

The next section provides a brief introduction to Extended ML. Section 3 reviews the notion of an institution and gives three examples. The new notion of an *institution with syntax* is then presented, and it is shown how the three example institutions can be extended to institutions with syntax. This is followed by a discussion of the additional assumptions which are necessary to allow signatures to be viewed as sets with structure, insofar as this is required by the semantics of Extended ML. The main ideas of the semantics itself are presented in section 4. As this presentation of the semantics glosses over some of the more complicated but uninteresting details, the fastidious reader may wish to refer to the full semantics which will appear separately as [ST 86].

2 Extended ML — an overview

The aim of this section is to review the main features of and motivations behind the Extended ML specification language in an attempt to make this paper self-contained. A more complete introduction to Extended ML is given in [ST 85a]. Although the examples below will contain bits of Standard ML code, the reader need not be acquainted with the features and syntactic details of Standard ML itself, especially since one of the goals of this paper is to make Extended ML entirely independent of Standard ML (although *not* of Standard ML's modularisation facilities, which we regard in this paper as separate from Standard ML itself). It will be sufficient to know that a collection of Standard ML declarations defines a set of types and values, where some values are functions and others are constants. A complete description of the language appears in [Mil 85].

Extended ML is based on the modularisation facilities for Standard ML proposed in [MacQ 85]. These facilities are designed to allow large Standard ML programs to be structured into modules with explicitly-specified interfaces. Under this proposal, interfaces (called *signatures*) and their implementations (called *structures*[3]) are defined separately. Every structure has a signature which gives the names of the types and values defined in the structure. Structures may be built on top of existing structures, so each one is actually a *hierarchy* of structures, and this is also reflected in its signature. Components of structures are accessed using qualified names such as `A.B.n` (referring to the component `n` of the structure component `B` of the structure `A`). *Functors*[4] are "parameterised" structures; the application of a functor to a structure yields a structure. (Contrary to category-theorists' expectations, there is no "morphism part"!) A functor has an input signature describing structures to which it may be applied, and an output signature describing the result of an application. A functor may have several parameters. It is possible, and sometimes necessary to allow interaction between different parts of a program, to declare that certain substructures (or just certain types) in the hierarchy are identical or *shared*.

An example of a simple program in Standard ML with modules is the following:

```
signature POSig =
   sig type elem
       val le : elem * elem -> bool
   end

signature SortSig =
   sig structure Elements : POSig
       val sort : Elements.elem list -> Elements.elem list
   end

functor Sort(PO : POSig) : SortSig =
   struct structure Elements = PO
          fun insert(a,nil) = a::nil
            | insert(a,b::l) = if Elements.le(a,b) then a::b::l
                                else b::insert(a,l)
          and sort nil = nil
            | sort(a::l) = insert(a,sort l)
   end

structure IntPO : POSig =
   struct type elem = int
          val le = op <=
   end

structure SortInt = Sort(IntPO)
```

Now, `SortInt.sort` may be applied to the list `[11,5,8]` to yield `[5,8,11]`.

In this example, the types of the values `sort` and `insert` in the functor `Sort` are inferred by the ML typechecker; the type of `sort` must be as declared in the signature `SortSig` while the value

---

[3] Structures were called *instances* in the 1984 version of [MacQ 85].

[4] Functors were called *modules* in the 1984 version of [MacQ 85].

`insert` is local to the definition of `Sort` since it is not mentioned in `SortSig`. Certain built-in types and values are *pervasive* — that is, they are implicitly a part of every signature and structure. In this example, the pervasive types `int` and `list` are used together with the pervasive values `nil`, `::` (add an element to the front of a list), and `<=` (i.e. $\leq$). The pervasive types and values may be regarded as forming a structure `Perv` with signature `PervSig` which is automatically included as an open substructure of every signature and structure ("open" means that a component n of `Perv` may be accessed using the name n rather than the name `Perv.`n). By the way, `list` is a so-called *polymorphic* type constructor, since it can be applied to any type `t` to form a type `t list`. The function `::` is a polymorphic function of type α `*` α `list ->` α `list`, meaning that it can be used to build lists with elements of any (uniform) type.

The information in a signature is sufficient for the use of Standard ML as a programming language, but when viewed as an interface specification a signature does not provide enough information to permit proving program correctness (for example). To make signatures more useful as interfaces of structures in program specification and development, one could allow them to include *axioms* which put constraints on the permitted behaviour of the components of the structure. An example of such a signature[5] is the following more informative version of the signature `POSig` above:

```
signature POSig =
   sig type elem
       val le : elem * elem -> bool
       axiom forall x:elem. le(x,x) = true
       and   forall x,y:elem. le(x,y) = true and le(y,x) = true => x=y
       and   forall x,y,z:elem.
                    le(x,y) = true and le(y,z) = true => le(x,z) = true
   end
```

This includes the previously-unexpressible precondition which `IntPO` must satisfy if `Sort(IntPO)` is to function as expected, namely that `IntPO.le` is a partial order on `IntPO.elem`.

Formal specifications can be viewed as abstract programs. Some specifications are so completely abstract that they give no hint of an algorithm (e.g. the specification of the inverse of a matrix A as that matrix A^{-1} such that $A \times A^{-1} = I$) and often it is not clear if an algorithm exists at all, while other specifications are so concrete that they amount to programs (e.g. Standard ML programs, which are just equations of a certain form which happen to be executable). In order to allow different stages in the evolution of a program to be expressed in the same framework, one could allow structures to contain a mixture of ML code and non-executable axioms. Functors could include axioms as well since they are simply parameterised structures. For example, a stage in the development of the functor `Sort` might be the following:

[5] We retain the term "signature" although this new version of `POSig` looks much more like a *theory* or *specification* than a *signature* (as these words are used in algebraic specification).

```
functor Sort(PO : POSig) : SortSig =
   struct structure Elements = PO
          val insert : Elements.elem * Elements.elem list -> Elements.elem list
          axiom forall a,a1,a2:Elements.elem, l,l1,l2:Elements.elem list.
                      insert(a,l) = l1@(a::l2)
                             => l1@l2 = l
                                and (member(a1,l1) => Elements.le(a1,a))
                                and (member(a2,l2) => Elements.le(a,a2))
          fun sort nil = nil
            | sort(a::l) = insert(a,sort l)
   end
```

(where `@` is the append function on lists). In this functor declaration, the function `sort` has been defined in an executable fashion in terms of `insert` which is so far only constrained by an axiom.

Extended ML is the result of extending the modularisation facilities of Standard ML as indicated above, that is by allowing axioms in signatures and in structures. Syntactically, the only significant change is to add the construct `axiom` *ax* to the list of alternative forms of *elementary specifications* (i.e. declarations allowed inside a signature body) and *elementary declarations* (declarations allowed inside a structure body). Signatures and structures both denote classes of algebras, where a signature Σ may be regarded as an interface for a structure S if the class of algebras associated with S is contained in the class associated with Σ. Functors are functions taking classes of algebras (contained in the class associated with its input signature) to classes of algebras (contained in the class associated with its output signature). The role of signatures as interfaces suggests that they should be regarded only as descriptions of the externally observable behaviour of structures. Thus, signatures should not distinguish between *behaviourally equivalent* algebras in which computations produce the same results of "external" types. Said another way, a signature identifies algebras which satisfy the same sentences from a pre-specified set Φ meant to describe the properties of structures which are externally observable, i.e. it describes a class of algebras which is closed under *observational equivalence* with respect to Φ (see [ST 85b] for an explanation of how this covers behavioural equivalence as a special case, more motivation for the use of this notion here, and much more technical detail). This is achieved in the semantics by first obtaining the class of algebras which "literally" satisfies the axioms of a signature and then *behaviourally abstracting* (closing under observational equivalence with respect to Φ) to obtain the class of algebras which "behaviourally" satisfies the axioms (cf. [Rei 84]).

In section 4 the semantics of Extended ML will be defined by translation into the ASL kernel specification language [SW 83], [Wir 83], [ST 85c]. From this semantic basis Extended ML inherits a formal notion of what it means for one specification to be an *implementation* or *refinement* of another. It also inherits *proof rules* which enable refinement steps to be proved correct. By composing verified refinement steps, it is possible to develop a guaranteed-correct program from a specification in a stepwise and modular fashion. An example of (part of) the development of an interpreter for a very simple programming language in Extended ML is given in [ST 85a].

3 Institutions with syntax

Any approach to formal specification must be based on some logical framework. The pioneering papers [GTW 76], [Gut 75], [Zil 74] used many-sorted equational logic for this purpose. Nowadays, however, examples of logical systems in use include first-order logic (with and without equality), Horn-clause logic, higher-order logic, infinitary logic, temporal logic and many others. Note that all these logical systems may be considered with or without predicates, admitting partial operations or not. This leads to different concepts of signature and of model; for example, to specify Standard ML programs we need polymorphic signatures and algebras with partial operations (so-called *partial algebras*), higher-order functions, etc.

The informal notion of a logical system has been formalised by Goguen and Burstall [GB 84], who introduced for this purpose the notion of an *institution*. An institution consists of a collection of signatures together with for any signature Σ a set of Σ-sentences, a collection of Σ-models and a satisfaction relation between Σ-models and Σ-sentences. Note that signatures are arbitrary abstract objects in this approach, not necessarily the usual "algebraic" signatures used in many standard approaches to algebraic specification (see e.g. [GTW 76]). The only "semantic" requirement is that when we change signatures, the induced translations of sentences and models preserve the satisfaction relation. This condition expresses the intended independence of the meaning of a specification from the actual notation. Formally:

Definition 1 *An institution* **INS** *consists of:*

- *a category* $\mathbf{Sign_{INS}}$ *(of signatures),*

- *a functor* $\mathbf{Sen_{INS}}\colon \mathbf{Sign_{INS}} \to \mathbf{Set}$ *(where* **Set** *is the category of all sets;* $\mathbf{Sen_{INS}}$ *gives for any signature* Σ *the set of* Σ*-sentences and for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$ *the function* $\mathbf{Sen_{INS}}(\sigma)\colon \mathbf{Sen_{INS}}(\Sigma) \to \mathbf{Sen_{INS}}(\Sigma')$ *translating* Σ*-sentences to* Σ'*-sentences),*

- *a functor* $\mathbf{Mod_{INS}}\colon \mathbf{Sign_{INS}} \to \mathbf{Cat}^{op}$ *(where* **Cat** *is the category of all categories;*[6] $\mathbf{Mod_{INS}}$ *gives for any signature* Σ *the category of* Σ*-models and for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$ *the* σ*-reduct functor* $\mathbf{Mod_{INS}}(\sigma)\colon \mathbf{Mod_{INS}}(\Sigma') \to \mathbf{Mod_{INS}}(\Sigma)$ *translating* Σ'*-models to* Σ*-models), and*

- *a satisfaction relation* $\models_{\Sigma,\mathbf{INS}} \subseteq |\mathbf{Mod_{INS}}(\Sigma)| \times \mathbf{Sen_{INS}}(\Sigma)$ *for each signature* Σ

such that for any signature morphism $\sigma\colon \Sigma \to \Sigma'$ *the translations* $\mathbf{Mod_{INS}}(\sigma)$ *of models and* $\mathbf{Sen_{INS}}(\sigma)$ *of sentences preserve the satisfaction relation, i.e. for any* $\varphi \in \mathbf{Sen_{INS}}(\Sigma)$ *and* $M' \in |\mathbf{Mod_{INS}}(\Sigma')|$

$$M' \models_{\Sigma',\mathbf{INS}} \mathbf{Sen_{INS}}(\sigma)(\varphi) \Longleftrightarrow \mathbf{Mod_{INS}}(\sigma)(M') \models_{\Sigma,\mathbf{INS}} \varphi \qquad \textit{(Satisfaction condition)}$$

[6]Of course, some foundational difficulties are connected with the use of this category, as discussed in [MacL 71]. We do not discuss this point here, and we disregard other such foundational issues in this paper; in particular, we sometimes use the term "collection" to denote a "set" which may be too large to really be a set.

The work of [Bar 74] on abstract model theory is similar in intent to the theory of institutions but the notions used and the conditions they must satisfy are more restrictive and rule out many of the examples we would like to deal with.

Notational conventions:

- The subscripts **INS** and Σ are omitted when there is no danger of confusion.
- For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, $\mathbf{Sen}(\sigma)$ is denoted just by σ and $\mathbf{Mod}(\sigma)$ is denoted by $\_|_\sigma$ (i.e. for $\varphi \in \mathbf{Sen}(\Sigma)$, $\sigma(\varphi)$ stands for $\mathbf{Sen}(\sigma)(\varphi)$ and for $M' \in |\mathbf{Mod}(\Sigma')|$, $M'|_\sigma$ stands for $\mathbf{Mod}(\sigma)(M')$).
- Satisfaction relations are extended to collections of models and sentences in the usual way.

Example 1 The institution **GEQ** of ground equations

An *algebraic signature* is a pair $\langle S, \Omega\rangle$ where S is a set (of sort names) and Ω is a family of mutually disjoint sets $\{\Omega_{w,s}\}_{w\in S^*, s\in S}$ (of operation names). We write $f\colon w \to s$ to denote $w \in S^*$, $s \in S$, $f \in \Omega_{w,s}$. An *algebraic signature morphism* $\sigma\colon \langle S,\Omega\rangle \to \langle S',\Omega'\rangle$ is a pair $\langle \sigma_{\mathrm{sorts}}, \sigma_{\mathrm{opns}}\rangle$ where $\sigma_{\mathrm{sorts}}\colon S \to S'$ and $\sigma_{\mathrm{opns}} = \{\sigma_{w,s}\colon \Omega_{w,s} \to \Omega'_{\sigma^*(w),\sigma(s)}\}_{w\in S^*, s\in S}$ is a family of maps where $\sigma^*(s_1,\ldots,s_n)$ denotes $\sigma_{\mathrm{sorts}}(s_1),\ldots,\sigma_{\mathrm{sorts}}(s_n)$ for $s_1,\ldots,s_n \in S$. We will write $\sigma(s)$ for $\sigma_{\mathrm{sorts}}(s)$, $\sigma(w)$ for $\sigma^*(w)$ and $\sigma(f)$ for $\sigma_{w,s}(f)$, where $f \in \Omega_{w,s}$.

The category of algebraic signatures **AlgSig** has algebraic signatures as objects and algebraic signature morphisms as morphisms; the composition of morphisms is the composition of their corresponding components as functions. (This obviously forms a category.)

Let $\Sigma = \langle S, \Omega\rangle$ be an algebraic signature.

A Σ-*algebra* A consists of an S-indexed family of carrier sets $|A| = \{|A|_s\}_{s\in S}$ and for each $f\colon s_1,\ldots,s_n \to s$ a function $f_A\colon |A|_{s_1} \times \cdots \times |A|_{s_n} \to |A|_s$. A Σ-*homomorphism* from a Σ-algebra A to a Σ-algebra B, $h\colon A \to B$, is a family of functions $\{h_s\colon |A|_s \to |B|_s\}_{s\in S}$ such that for any $f\colon s_1,\ldots,s_n \to s$ and $a_1 \in |A|_{s_1},\ldots,a_n \in |A|_{s_n}$, $h_s(f_A(a_1,\ldots,a_n)) = f_B(h_{s_1}(a_1),\ldots,h_{s_n}(a_n))$.

The category of Σ-algebras $\mathbf{Alg}(\Sigma)$ has Σ-algebras as objects and Σ-homomorphisms as morphisms; the composition of homomorphisms is the composition of their corresponding components as functions. (This obviously forms a category.)

For any algebraic signature morphism $\sigma\colon \Sigma \to \Sigma'$ and Σ'-algebra A', the σ-*reduct* of A' is the Σ-algebra $A'|_\sigma$ such that $|A'|_\sigma|_s = |A|_{\sigma(s)}$ for $s \in S$ and $f_{A'|_\sigma} = \sigma(f)_{A'}$ for $f\colon w \to s$ in Σ. For a Σ'-homomorphism $h'\colon A' \to B'$ where A' and B' are Σ'-algebras, the σ-reduct of h' is the Σ-homomorphism $h'|_\sigma\colon A'|_\sigma \to B'|_\sigma$ defined analogously. The mappings $A' \mapsto A'|_\sigma$, $h' \mapsto h'|_\sigma$ form a functor from $\mathbf{Alg}(\Sigma')$ to $\mathbf{Alg}(\Sigma)$.

For any algebraic signature Σ, $\mathbf{Alg}(\Sigma)$ contains an initial object T_Σ which is (to within isomorphism) the algebra of ground Σ-terms (see e.g. [GTW 76]). A *ground* Σ-*equation* is a pair $\langle t, t'\rangle$ (usually written as $t = t'$) where t, t' are ground Σ-terms of the same sort.

By definition, for any Σ-algebra A there is a unique Σ-homomorphism $h\colon T_\Sigma \to A$. For any ground term $t \in |T_\Sigma|_s$ (for s in the sorts of Σ) we write t_A rather than $h_s(t)$ to denote the value of t in A. For any Σ-algebra A and ground Σ-equation $t = t'$ we say that $t = t'$ *holds* in A (or A *satisfies* $t = t'$), written $A \models t = t'$, if $t_A = t'_A$.

Let $\sigma: \Sigma \to \Sigma'$ be an algebraic signature morphism. The unique Σ-homomorphism $h: T_\Sigma \to T_{\Sigma'}|_\sigma$ determines a translation of Σ-terms of Σ'-terms. For a ground Σ-term t of sort s we write $\sigma(t)$ rather than $h_s(t)$. This in turn determines a translation (again denoted by σ) of ground Σ-equations to ground Σ'-equations: $\sigma(t = t') =_{def} \sigma(t) = \sigma(t')$.

All of the above notions combine to form the institution of ground equations **GEQ**:

- $\mathbf{Sign}_{\mathbf{GEQ}}$ is the category of algebraic signatures **AlgSig**.
- For an algebraic signature Σ, $\mathbf{Sen}_{\mathbf{GEQ}}(\Sigma)$ is the set of all ground Σ-equations; for an algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Sen}_{\mathbf{GEQ}}(\sigma)$ maps any ground Σ-equation $t = t'$ to the ground Σ'-equation $\sigma(t) = \sigma(t')$.
- For an algebraic signature Σ, $\mathbf{Mod}_{\mathbf{GEQ}}(\Sigma)$ is $\mathbf{Alg}(\Sigma)$; for an algebraic signature morphism $\sigma: \Sigma \to \Sigma'$, $\mathbf{Mod}_{\mathbf{GEQ}}(\sigma)$ is the functor $\_|_\sigma: \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$.
- For an algebraic signature Σ, $\models_{\Sigma,\mathbf{GEQ}}$ is the satisfaction relation as defined above.

It is easy to check that **GEQ** is an institution (the satisfaction condition is a special case of the Satisfaction Lemma of [BG 80]).

We have presented the above example in such detail to show explicitly how standard definitions may be put together to fit into the mould of an institution. The following examples will be presented in a more sketchy way.

Example 2 The institution **OSGEQ** of ground equations in order-sorted algebras

This example is based on [GJM 85] where the reader may find a more detailed presentation, a number of interesting technical results and examples indicating how this institution may be used in algebraic specification.

An *order-sorted signature* is a triple $\langle S, \le, \Omega\rangle$ where $\langle S, \le\rangle$ is a partially ordered set (of sort names) and $\Omega = \{\Omega_{w,s}\}_{w\in S^*, s\in S}$ is a family of sets (of operation names). If $s \le s'$ for $s, s' \in S$, we say that s is a *subsort* of s'. Thus, roughly, an order-sorted signature is an algebraic signature with a subsort ordering on sorts. We extend the subsort ordering to lists of sorts of the same length in the usual (component-wise) way. We also use the same notational conventions as in the previous example for algebraic signatures. Unlike the case of algebraic signatures, however, we do not require the sets $\{\Omega_{w,s}\}_{w\in S^*, s\in S}$ to be mutually disjoint. On the contrary, we assume that if $f: w \to s$ is an operation name then $f: w' \to s'$ is also an operation name for any $w' \le w$ and $s \le s'$. Moreover, for technical reasons (cf. [GJM 85]) we assume that order-sorted signatures are *regular*, i.e. for any $w^* \in S^*$, if $f: w \to s$ for some $w^* \le w$ then there is a least $\langle w', s'\rangle \in S^* \times S$ such that $w^* \le w'$ and $f: w' \to s'$.

Order-sorted signature morphisms are defined in the same way as algebraic signature morphisms, except that additionally their sort components are required to be monotonic with respect to the subsort orderings and their operation-name components to preserve the identity of operation names with different arities and result sorts.

This defines the category **OrdSig** of (regular) order-sorted signatures and their morphisms, which is the category of signatures in the institution **OSGEQ**.

Let $\Sigma = \langle S, \le, \Omega\rangle$ be an order-sorted signature.

An *order-sorted* Σ*-algebra* is just a $\langle S, \Omega\rangle$-algebra A (in the sense of the previous example) such that $|A|_s \subseteq |A|_{s'}$ whenever $s \leq s'$ in Σ, and such that for $f: w \to s$, $w' \leq w$ and $s \leq s'$, the function corresponding to $f: w' \to s'$ in A is the set-theoretic restriction of the function corresponding to $f: w \to s$. Similarly, an *order-sorted* Σ*-homomorphism* is just a $\langle S, \Omega\rangle$-homomorphism h such that h_s is a restriction of $h_{s'}$ if $s \leq s'$. This defines the category $\mathbf{OSAlg}(\Sigma)$ of order-sorted Σ-algebras, which is the category of Σ-models in **OSGEQ**. For any order-sorted signature morphism $\sigma: \Sigma \to \Sigma'$, we define the σ*-reduct functor* $\_|_\sigma: \mathbf{OSAlg}(\Sigma') \to \mathbf{OSAlg}(\Sigma)$ exactly as in **GEQ**.

Finally, the sentences of **OSGEQ** and their satisfaction by order-sorted algebras is defined in the same way as in the standard algebraic case, i.e. in **GEQ**. Notice, however, that in the order-sorted case terms may have more than one sort. It turns out (see [GJM 85]) that every term built over a regular order-sorted signature has a least (most general) sort. Moreover, the value of a term in an order-sorted algebra does not depend on which of its sorts we consider. Thus, whether a ground equation $t = t'$ is satisfied by an order-sorted algebra or not is independent of which of the common sorts of t and t' we choose to evaluate the terms in.

Please note that the above examples deal with ground equations for the sake of simplicity of exposition rather than because of any inability of the notion of institution to cope with variables. For example, an institution **FOEQ** may be obtained from **GEQ** by changing the definition of the **Sen** functor so that when applied to $\Sigma \in |\mathbf{AlgSig}|$ it produces the set of all sentences of first-order predicate logic over Σ with equality (the satisfaction relation must be modified accordingly). Another institution may be obtained from **OSGEQ** in the same fashion. There are in fact general constructions for introducing variables into the sentences of an arbitrary institution and for binding such variables with different quantifiers (see [ST 85c], [Tar 84]) as well as for introducing logical connectives (cf. [Bar 74]), so **FOEQ** can be produced from **GEQ** by "iterative" application of a sequence of general constructions.

Example 3 The institution **CRCAT** of commutativity requirements in categories

The third example we consider is of a slightly non-standard character. We present a simple logic for stating that certain diagrams in a category commute. We will consider categories with named objects and morphisms. Sentences of the logical system we describe will allow one to require that morphisms produced by composition of series of (named) morphisms coincide.

The category of signatures in **CRCAT** is the category **Graph** of *directed graphs*, i.e. the category of algebras over the algebraic signature having the two sorts *node* and *edge* and the two operations *source*: *edge* $\to$ *node* and *target*: *edge* $\to$ *node*, together with their (homo)morphisms.

Then, a model over a graph G is a category $\mathbf{C}$ having a subcategory with "shape" G, i.e. a graph morphism $F: G \to C$ from G to the underlying graph (the pre-category) of $\mathbf{C}$.

For two G-models $F1: G \to C1$ and $F2: G \to C2$, a G*-morphism* in $\mathbf{Mod}_{\mathbf{CRCAT}}(G)$ from $F1$ to $F2$ is a functor $\mathbf{F}: \mathbf{C1} \to \mathbf{C2}$ such that $F1;\mathbf{F} = F2$.

For any graph G and two nodes $s, t \in |G|_{nodes}$, a *path* in G with source s and target t (or, from s to t) is a sequence $e_1 \ldots e_n$ $(n \geq 1)$ of edges in G such that $source_G(e_1) = s$, $target_G(e_n) = t$ and $source_G(e_i) = target_G(e_{i-1})$ for $1 < i \leq n$. Moreover, for each node s we have the *empty path* ε_s (from s to s).

For any G-model $F: G \to C$, a path p from s to t in G determines a morphism $F(p): F(s) \to F(t)$ in $\mathbf{C}$ defined by $F(\varepsilon_s) = id_{F(s)}$ for $s \in |G|_{nodes}$ and $F(e_1 \ldots e_n) = F(e_1); \cdots ;F(e_n)$ for a non-empty path $e_1 \ldots e_n$ in G.

(The above definitions may be formulated in a much more compact way using the standard machinery of category theory. Namely, for any graph G, the category of G-models has an initial model, which is — up to isomorphism — the category of paths in G. The evaluation $p \mapsto F(p)$ is given by the unique G-morphism — a functor — from this initial G-model to $F\colon G \to C$.)

By a path equation in G we mean any pair of paths in G with the same sources and targets, respectively. We say that a G-model $F\colon G \to C$ *satisfies* a path equation $\langle p, q\rangle$ if $F(p) = F(q)$.

Finally, for any graph G (a signature in $\mathbf{Sign}_{\mathbf{CRCAT}}$) G-sentences in **CRCAT** are sets of path equations in G, where a G-model satisfies a G-sentence Φ if it satisfies all path equations $\varphi \in \Phi$.

As mentioned before, the notion of an institution is not sufficiently rich to deal with all the syntactic-level details of the Extended ML semantics we want to describe. We need to associate with every signature of the underlying institution the vocabulary of names it provides. Then, we assume that the sentences over any signature have "syntactic representations" built using the vocabulary of names provided by the signature. We require that every signature morphism $\sigma\colon \Sigma \to \Sigma'$ induces a translation of the names of Σ to the names of Σ'. This in turn will induce a translation between the syntactic representations of Σ-sentences and Σ'-sentences, which must be compatible with the translation of sentences in the underlying institution. All this leads to the following definition:

Definition 2 *An institution with syntax is an institution* $\mathbf{INS} = \langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models_\Sigma\}_{\Sigma \in |\mathbf{Sign}|}\rangle$ *together with:*

- *a functor* $\mathbf{Names}\colon \mathbf{Sign} \to \mathbf{Set}$ *(which gives the vocabulary of names a signature provides),*
- *an endofunctor* $\mathbf{Syn}\colon \mathbf{Set} \to \mathbf{Set}$ *(which gives the set of syntactic representations of sentences over a vocabulary of names),*
- *for any signature* $\Sigma \in |\mathbf{Sign}|$*, a partial function* $repr_\Sigma\colon \mathbf{Syn}(\mathbf{Names}(\Sigma)) \rightharpoonup \mathbf{Sen}(\Sigma)$ *(which associates syntactic representations with sentences)*

such that:

1. *the functor* **Names** *is faithful, i.e. for any two "parallel" signature morphisms* $\sigma_1, \sigma_2\colon \Sigma \to \Sigma'$*, if* $\mathbf{Names}(\sigma_1) = \mathbf{Names}(\sigma_2)$ *then* $\sigma_1 = \sigma_2$*;*

2. *any syntactic representation of a sentence determines the set of names it actually uses, i.e. for any set* N *and* $ax \in \mathbf{Syn}(N)$*, there is a least set* $N' \subseteq N$ *such that* $ax \in \mathbf{Syn}(N')$*;*

3. *the family of representation functions is a natural transformation* $repr\colon \mathbf{Names};\mathbf{Syn} \dashrightarrow \mathbf{Sen}$ *(in the category of sets with partial functions), i.e. for any signature morphism* $\sigma\colon \Sigma \to \Sigma'$ *we have* $\mathbf{Syn}(\mathbf{Names}(\sigma));repr_{\Sigma'} = repr_\Sigma;\mathbf{Sen}(\sigma)$ *(as partial functions), which is to say that the following diagram commutes:*

$$\begin{tikzcd}
\mathbf{Syn}(\mathbf{Names}(\Sigma)) \arrow[r, "repr_\Sigma"] \arrow[d, "\mathbf{Syn}(\mathbf{Names}(\sigma))"'] & \mathbf{Sen}(\Sigma) \arrow[d, "\mathbf{Sen}(\sigma)"] \\
\mathbf{Syn}(\mathbf{Names}(\Sigma')) \arrow[r, "repr_{\Sigma'}"'] & \mathbf{Sen}(\Sigma')
\end{tikzcd}$$

The intuition behind the representation functions $\{repr_\Sigma\}_{\Sigma\in|\mathbf{Sign}|}$ is that computing the value of such a function on an argument corresponds to the parsing and typechecking of (a representation of) a sentence. Notice that for any signature Σ, $repr_\Sigma$ is only a *partial* function. There may be objects in $\mathbf{Syn}(\mathbf{Names}(\Sigma))$ which actually do not represent any sentence from $\mathbf{Sen}(\Sigma)$. This corresponds, for example, to the case of ill-typed equations (see examples below). We do not require $repr_\Sigma$ to be surjective either; this allows some sentences to have no syntactic representation. For example, if $\mathbf{Sen}(\Sigma)$ is some class of infinitary formulae, it might be possible to find syntactic representations for some sentences (e.g. the finitely presentable ones) but not for others. Alternatively, suppose $\mathbf{Sen}(\Sigma)$ includes data constraints [GB 84]; one would expect those involving theories with no recursively enumerable axiomatization to have no syntactic representation. Furthermore, we do not require $repr_\Sigma$ to be injective, since a sentence might have multiple syntactic representations. To take a simple example, we could allow redundant parentheses to be inserted into syntactic representations of first-order formulae. We *do* require $repr_\Sigma$ to be a function, which means that representations are unambiguous, i.e. any syntactic object over the vocabulary of names of a signature represents at most one sentence over this signature.

The recent work of Goguen and Burstall [GB 86] on charters and parchments opens similar possibilities as our notion of an institution with syntax does (among others). In their "parchment chartered institutions", however, sentences just *are* (rather than *are represented by*, as in our institutions with syntax) well-formed finitary syntactic objects, which excludes some of the cases mentioned above. It may be interesting to check if our semantics of Extended ML may be given in the framework of parchments (with some additional assumptions corresponding to our assumptions about the functor **Names**).

Example 1 The institution **GEQ** of ground equations (revisited)

There is a natural way to extend the institution **GEQ** to an institution with syntax. Namely, for any algebraic signature $\Sigma = \langle S, \Omega\rangle$, let $\mathbf{Names_{GEQ}}(\Sigma)$ be the disjoint union of S and (the union of) Ω. Then, for any set N, let $\mathbf{Syn_{GEQ}}(N)$ be the set of all pairs of (any representation of) finite trees with nodes labelled by elements of N. Both of these extend to functors in the obvious way. Finally, for any algebraic signature Σ and pair of labelled trees $\langle t, t'\rangle \in \mathbf{Syn}(\mathbf{Names}(\Sigma))$, to compute $repr_{\Sigma,\mathbf{GEQ}}(\langle t, t'\rangle)$ one has to check whether t and t' are syntactically well-formed and well-typed Σ-terms of the same sort. If this is the case, the result is the obvious equation in $\mathbf{Sen_{GEQ}}(\Sigma)$; otherwise the result is undefined. It is easy to check that these extensions to **GEQ** satisfy the required conditions.

Example 2 The institution **OSGEQ** of ground equations in order-sorted algebras (revisited)

The institution **OSGEQ** may be extended to an institution with syntax in exactly the same way as the institution **GEQ** was in the previous example. The only difference is that the typechecking of terms is more elaborate in the order-sorted case (cf. [GJM 85]).

Example 3 The institution **CRCAT** of commutativity requirements in categories (revisited)

Again, the definition of the vocabulary of names used in a graph G (a signature in $|\mathbf{Sign_{CRCAT}}|$) is obvious: $\mathbf{Names_{CRCAT}}(G)$ is the disjoint union of the sets of nodes and of edges in G. Then, for any set N, $\mathbf{Syn_{CRCAT}}(N)$ is the set of all (isomorphism classes of) finite directed graphs with nodes and edges labelled by elements of N. Again, both of these extend in the obvious way to functors.

For any graph $G \in |\mathbf{Sign}_{\mathbf{CRCAT}}|$ and graph $D \in \mathbf{Syn}_{\mathbf{CRCAT}}(\mathbf{Names}_{\mathbf{CRCAT}}(G))$, to compute $repr_{G,\mathbf{CRCAT}}(D)$ we first check whether the labelling of D is consistent with G, i.e. whether the nodes of D are labelled by nodes of G and whether the edges of D are labelled by edges of G with source and target nodes as in G. If this is not the case, the result is undefined. Otherwise, $repr_{G,\mathbf{CRCAT}}(D)$ is the set of path equations over G corresponding — by taking sequences of edge labels instead of sequences of edges — to the set of pairs of finite (and possibly empty) paths in D with the same sources and targets, respectively.

In other words, $repr_{G,\mathbf{CRCAT}}(D)$ is a set of path equations semantically equivalent to the requirement that D is a commutative diagram.[7] Note that in contrast to the two previous examples, the representation functions in **CRCAT** are neither injective nor surjective in general.

In the rest of this section we will explore the possibilities opened by our having enriched the structure of the category **Sign** in an institution with syntax to form a *concrete category* $\langle\mathbf{Sign},\mathbf{Names}\rangle$ (see [MacL 71]). The idea is that we want to "lift" some basic set-theoretic notions from the sets of names in signatures to signatures themselves, i.e. from the category **Set** to the (concrete) category **Sign**. We will need some additional assumptions, used also later in the definition of the semantics of Extended ML. (What follows works in an arbitrary concrete category and may in fact be well-known folklore in the theory of concrete categories, but we were not able to locate appropriate references.)

We say that a signature morphism $\iota\colon \Sigma \to \Sigma'$ is a *signature inclusion* if $\mathbf{Names}(\iota)$ is an inclusion (of $\mathbf{Names}(\Sigma)$ into $\mathbf{Names}(\Sigma')$). If there exists a signature inclusion from Σ to Σ', we call Σ a *subsignature* of Σ'. Notice that then the signature inclusion is unique, since the functor **Names** is faithful; we denote it by $\iota_{\Sigma\subseteq\Sigma'}$.

A subsignature Σ of Σ' is said to be *full* if every subsignature of Σ' with the same set of names as Σ is a subsignature of Σ. Notice that in $\langle\mathbf{AlgSig},\mathbf{Names}_{\mathbf{GEQ}}\rangle$ and $\langle\mathbf{Graph},\mathbf{Names}_{\mathbf{CRCAT}}\rangle$ the notions of subsignature and of full subsignature coincide — every subsignature is full. This is not the case in $\langle\mathbf{OrdSig},\mathbf{Names}_{\mathbf{OSGEQ}}\rangle$, though. An order-sorted subsignature Σ of Σ' is full if and only if Σ inherits from Σ' all the subsort requirements concerning the sorts of Σ.

We call a set of names $N \subseteq \mathbf{Names}(\Sigma)$ *closed* in Σ if there is a subsignature Σ' of Σ with the set of names N, i.e. such that $\mathbf{Names}(\Sigma') = N$.

For any set $N \subseteq \mathbf{Names}(\Sigma)$, a *signature generated in Σ by N* is a full subsignature Σ' of Σ such that $\mathbf{Names}(\Sigma')$ is the smallest set containing N and closed in Σ.

Assumption 1 *For any signature Σ and set $N \subseteq \mathbf{Names}(\Sigma)$ there exists a unique signature generated by N in Σ, denoted $\Sigma|_N$.*

Corollary *If $\iota\colon \Sigma \to \Sigma'$ is an isomorphism such that $\mathbf{Names}(\iota)$ is an identity (on $\mathbf{Names}(\Sigma) = \mathbf{Names}(\Sigma')$) then ι is itself an identity (on $\Sigma = \Sigma'$).*

Proof *By definition we have $\Sigma = \Sigma'|_{\mathbf{Names}(\Sigma')} = \Sigma'$. Then, since the functor **Names** is faithful (and, of course, preserves identities) ι must be the identity morphism.*

---

[7] The definitions could be altered slightly to more accurately reflect normal usage, so that (for example) the diagram $A \xrightarrow{f} B \overset{g}{\underset{h}{\rightrightarrows}} C$ would require that $f;g = f;h$ but not that $g = h$. The corresponding diagram in our present version of **CRCAT** is a square with two copies of B and f.

Let $\mathcal{S}$ be a set of signatures, X a set and $\Phi = \{\varphi_\Sigma \colon \mathbf{Names}(\Sigma) \to X\}_{\Sigma\in\mathcal{S}}$ a family of functions. We say that $\mathcal{S}$ is *compatible along* Φ if there exists a signature Σ_{big} such that $X \subseteq \mathbf{Names}(\Sigma_{big})$ (let $in \colon X \hookrightarrow \mathbf{Names}(\Sigma_{big})$ be the inclusion) and a family of signature morphisms $\{\sigma_\Sigma \colon \Sigma \to \Sigma_{big}\}_{\Sigma\in\mathcal{S}}$ such that $\mathbf{Names}(\sigma_\Sigma) = \varphi_\Sigma;in$ for $\Sigma \in \mathcal{S}$. If this is the case, we say that Σ_{big} *contains* $\mathcal{S}$ *via* $\{\sigma_\Sigma \colon \Sigma \to \Sigma_{big}\}_{\Sigma\in\mathcal{S}}$. The signature induced by $\mathcal{S}$ along Φ is, intuitively, the least such signature Σ_{big}. Formally, we say that Σ_{ind} together with a family of signature morphisms $\{\sigma_\Sigma \colon \Sigma \to \Sigma_{ind}\}_{\Sigma\in\mathcal{S}}$ is *induced by* $\mathcal{S}$ *along* Φ if:

- Σ_{ind} contains $\mathcal{S}$ via $\{\sigma_\Sigma \colon \Sigma \to \Sigma_{ind}\}_{\Sigma\in\mathcal{S}}$;
- $\mathbf{Names}(\Sigma_{ind}) = X$; and
- Σ_{ind} is a subsignature of any signature Σ_{big} which contains $\mathcal{S}$ via $\{\sigma'_\Sigma \colon \Sigma \to \Sigma_{big}\}_{\Sigma\in\mathcal{S}}$.

Notice that if such a signature Σ_{ind} exists (together with such a family of signature morphisms) then:

1. Σ_{ind} is unique;
2. For each $\Sigma \in \mathcal{S}$, the signature morphism $\sigma_\Sigma \colon \Sigma \to \Sigma_{ind}$ is unique (σ_Σ will be denoted $\widehat{\varphi}_\Sigma$ in the sequel); and
3. For any Σ_{big} with $\{\sigma'_\Sigma \colon \Sigma \to \Sigma_{big}\}_{\Sigma\in\mathcal{S}}$ as above, the "universal inclusion" $\iota \colon \Sigma_{ind} \to \Sigma_{big}$ is unique and $\sigma_\Sigma;\iota = \sigma'_\Sigma$ for $\Sigma \in \mathcal{S}$.

Proof *Parts 2 and 3 follow directly from the faithfulness of* $\mathbf{Names}$. *For 1, suppose that both* Σ_{ind} *and* Σ'_{ind} *are induced by* $\mathcal{S}$ *along* Φ. *Then there exist signature inclusions* $\iota \colon \Sigma_{ind} \to \Sigma'_{ind}$ *and* $\iota' \colon \Sigma'_{ind} \to \Sigma_{ind}$. *Hence, both* $\iota;\iota'$ *and* $\iota';\iota$ *are signature inclusions, and hence identities (since* $\mathbf{Names}(\Sigma_{ind}) = X = \mathbf{Names}(\Sigma'_{ind})$*). Thus,* ι *is an isomorphism and* $\mathbf{Names}(\iota)$ *is an identity, which implies (by the corollary to assumption 1) that* $\Sigma_{ind} = \Sigma'_{ind}$.

Of course, in general Σ_{ind} does not have to exist. Even in the most standard case, for algebraic signatures, the role of each element of X in the induced signature must be identified. On a more formal level, this amounts to the requirement that the family Φ is (collectively) surjective, i.e. for every $x \in X$ there exists some $\Sigma \in \mathcal{S}$ and $n \in \mathbf{Names}(\Sigma)$ such that $\varphi_\Sigma(n) = x$.

Assumption 2 *For any family of signatures* $\mathcal{S} \subseteq |\mathbf{Sign}|$ *compatible along a surjective family of functions* $\Phi = \{\varphi_\Sigma \colon \mathbf{Names}(\Sigma) \to X\}_{\Sigma\in\mathcal{S}}$ *there exists a signature induced by* $\mathcal{S}$ *along* Φ *(which is then unique, by the above remarks).*

Let $\mathcal{S}$ be a family of signatures, $N = \bigcup_{\Sigma\in\mathcal{S}} \mathbf{Names}(\Sigma)$, and let $\mathcal{I} = \{in_\Sigma \colon \mathbf{Names}(\Sigma) \hookrightarrow N\}_{\Sigma\in\mathcal{S}}$ be the family of inclusions.

We say that $\mathcal{S}$ is *compatible* if it is compatible along $\mathcal{I}$. By the *union* of $\mathcal{S}$, written $\bigcup\mathcal{S}$, we mean the signature induced by $\mathcal{S}$ along $\mathcal{I}$. (If $\mathcal{S}$ is finite we may use the usual infix notation for the union.) We say that a signature Σ is *compatible with* $\mathcal{S}$ if the family $\mathcal{S} \cup \{\Sigma\}$ is compatible.

Lemma 1 *Let* $\mathcal{S}$ *be a compatible set of signatures.*

1. *Any subset* $\mathcal{S}' \subset \mathcal{S}$ *is compatible;*

2. *If Σ is a subsignature of a signature in $\mathcal{S}$ then Σ is compatible with $\mathcal{S}$; and*

3. *For any subset $\mathcal{S}' \subseteq \mathcal{S}$, $\bigcup \mathcal{S}'$ is compatible with $\mathcal{S}$.*

Proof *Obvious, since any signature which contains $\mathcal{S}$ also contains: (1) $\mathcal{S}'$, (2) Σ, and (3) $\bigcup \mathcal{S}'$.*

Intuitively, conflicts between signatures arise (signatures are incompatible) when different signatures use the same names in different ways. We formalise this intuition as follows:

Assumption 3 *Any finite family of signatures with disjoint names apart from a common full subsignature is compatible; i.e. for any finite family $\mathcal{S}$ of signatures, if there is a signature Σ_{perv} which is a full subsignature of each signature in $\mathcal{S}$ such that the sets* $\mathbf{Names}(\Sigma) - \mathbf{Names}(\Sigma_{perv})$ *are disjoint for different $\Sigma \in \mathcal{S}$, then $\mathcal{S}$ is compatible.*

4 The semantics of Extended ML

In this section we outline the main ideas behind the semantics of Extended ML (EML). Although some of the purely technical issues are not discussed here, all of the non-standard aspects of the semantics are treated. The reader who is interested in the full details of the semantics of Extended ML is referred to [ST 86].

As discussed in the introduction, the ideas presented here and the details in [ST 86] are independent of the institution with syntax in which the user of the language may choose to work. This means that we actually describe a family of specification languages, each one obtained by instantiating the definitions we provide in a given institution with syntax (together with some low-level details — see below). The examples in section 2 are written in the variant of Extended ML obtained by instantiating the definitions in a version of the institution **FOEQ** in which "executable" axioms may be written in Standard ML syntax to distinguish them from non-executable ones. We will use the term "the standard variant" to refer to this language below.

Throughout this section we assume that we are given an arbitrary but fixed institution with syntax, $\mathbf{INS} = \langle \mathbf{Sign}, \mathbf{Set}, \mathbf{Mod}, \models, \mathbf{Names}, \mathbf{Syn}, repr \rangle$. To clarify the overloaded term "signature" (sorry, but it's not our fault!) we adopt the convention that "signature" refers to signatures of **INS** (objects of **Sign**), denoted by Σ, Σ' etc. while "EML signature" refers to the concept of signature appearing in the specification language. For consistency, we will also use the terms "EML structure" and "EML functor".

The semantics of Extended ML we present is based on the ASL kernel specification language described in the framework of an arbitrary institution in [ST 85c]. Informally in this section (and formally in [ST 86]) we refer to the following specification-building operations of ASL:

- Form the *union* of a family of Σ-specifications $\{SP_i\}_{i \in I}$, specifying the collection of Σ-models satisfying SP_i for all $i \in I$. We sometimes speak of the *intersection* of model classes instead.

- *Translate* a Σ-specification to another signature Σ' along a signature morphism $\sigma: \Sigma \to \Sigma'$. This together with union allows large specifications to be built from smaller and more or less independent specifications.

- *Derive* a Σ'-specification from a specification over a richer signature Σ using a signature morphism $\sigma\colon \Sigma' \to \Sigma$. This makes it possible to forget or rename components of a specification while essentially preserving its collection of models.

- *Abstract* away from certain details of a specification, admitting any models which are observationally equivalent to a model of the specification with respect to some given set of properties (defined using sentences of the institution).

These operations can be viewed as functions on classes of models over a given signature.

The basic entities of Extended ML are EML signatures, structures and functors. We discuss each of these in turn.

4.1 EML signatures

As indicated before, an EML signature essentially denotes a class of models over a given signature Σ. However, we inherit from Standard ML certain complications which force us to adopt a more complex view. The most obvious one is that in Standard ML the same object may have multiple names which are said to *share*. The way we adopt here to cope with this is to assume that the names which occur in Σ are unique internal semantic-level names which are associated with one or more external identifiers to which the user may refer. Thus, the denotation of an EML signature is taken to be a quadruple $\langle N, \tau, \Sigma, C\rangle$ where:

- N is a set of (external) identifiers,

- Σ is a signature,

- $\tau\colon N \to \mathbf{Names}(\Sigma)$ is a function assigning an internal name to each external identifier, and

- $C \subseteq |\mathbf{Mod}(\Sigma)|$ is a class of Σ-models.

We call quadruples of this form *EML values*. We occasionally identify EML signatures (and structures) with the EML values they denote. Sharing is determined by identity of internal names, i.e. if $n, m \in N$ are two external identifiers then n and m share iff $\tau(n) = \tau(m)$. Every EML signature is closed in the sense that all components of its underlying signature Σ have associated external identifiers in N, i.e. τ is surjective. Substructures are not handled by maintaining an explicit hierarchy of EML values. Rather, this is done at the level of external identifiers by using identifiers like $A.n$ for the component n of the substructure A; any identifier having this form is assumed to refer to a component of A.

As in the standard variant, every EML signature contains the pervasive EML signature. This is modelled by the implicit assumption that every EML value which is generated as the denotation of an EML signature contains as a full subvalue the pervasive EML value $\langle N_{perv}, \tau_{perv}, \Sigma_{perv}, C_{perv}\rangle$, i.e. for any EML value $\langle N, \tau, \Sigma, C\rangle$, $N_{perv} \subseteq N$, $\tau\restriction N_{perv} = \tau_{perv}$, Σ_{perv} is a full subsignature of Σ and the Σ_{perv}-models derived from C using the signature inclusion $\iota_{\Sigma_{perv}\subseteq\Sigma}$ are included in C_{perv}. With this in mind we adopt the convention that when we say that two sets of names are disjoint, we mean that they are disjoint apart from pervasive names. Of course, the pervasive value depends on **INS** and must be provided when Extended ML is instantiated in a given institution.

Thus we start elaboration of an EML signature *sig* by including in its denotation the pervasive EML signature. This is gradually extended with each elementary specification in *sig*.

The most basic kind of elementary specification is to extend the underlying signature of the current EML value (in the standard variant, by adding new types/values). Again, this is one of the things which cannot be treated in a completely institution-independent way; we must assume that a signature morphism describing the extension is given. After all, the way such extensions arise depends fundamentally on what signatures really are and what structure is available on them. We assume, however, that this morphism never changes the existing names of the signature.

Thus, as a replacement for Standard ML syntax like `val f:t->t` and `type t`, Extended ML contains a construct `extend along` $\iota\colon \Sigma \to \Sigma'$ where Σ is the underlying signature of the current EML value and ι is a signature inclusion. We assume that some syntax analogous to `val f:t->t` and `type t` for describing such signature inclusions is provided when Extended ML is instantiated in a given institution. We will regard the Standard ML declaration `val v:t =` *exp* as an abbreviation for `val v:t; axiom v =` *exp*, and `fun f(`pat_1`:t):t' =` exp_1 `| ... | f(`pat_n`:t):t' =` exp_n as an abbreviation for `val f:t->t'; axiom f(`pat_1`) =` exp_1 `and ... and f(`pat_n`) =` exp_n. However, the declaration `type t = t'` is an abbreviation for `new name t for t'` (see below).

Note that ι may not only introduce new names; it also could enrich the structure of the signature Σ. For example, in order-sorted logic Σ may be the signature with two sorts $\mathtt{t}, \mathtt{t}'$ and Σ' could be Σ together with the requirement that $\mathtt{t} \leq \mathtt{t}'$ (and possibly some new names as well).

The semantics of this construct is straightforward: we simply change the current underlying signature to Σ', translate the class of models along ι, and make the new internal names $\mathbf{Names}(\Sigma') - \mathbf{Names}(\Sigma)$ available as external identifiers.

To simulate a different kind of elementary specification (e.g. `type t = t'` in the standard variant) which introduces a new external identifier for an existing internal object, Extended ML provides the construct `new name` n `for` m where n is a new external identifier and m is required to be an external identifier available in the current EML value. This construct just introduces the new external identifier n and binds it to the internal name associated with m.

Note that this construct already introduces the possibility of sharing by allowing a new external identifier to be bound to an existing object. Two existing external identifiers n and m can be forced to share by the elementary specification `sharing` n `=` m. The intuitive meaning of this is that nothing happens if n and m share already; otherwise the resulting underlying signature will identify the internal names associated with n and m. To describe the nontrivial case formally, consider a function $\varphi\colon \mathbf{Names}(\Sigma) \to (\mathbf{Names}(\Sigma) - \{\tau(n), \tau(m)\} \cup \{new\})$ where Σ is the underlying signature of the current EML value and *new* is an arbitrary name not in $\mathbf{Names}(\Sigma) - \{\tau(n), \tau(m)\}$, such that:

$$\varphi(x) = \begin{cases} new & \text{if } x \in \{\tau(n), \tau(m)\} \\ x & \text{otherwise} \end{cases}$$

(A little more care is necessary if either $\tau(n)$ or $\tau(m)$ is pervasive.) The resulting underlying signature is the signature induced by Σ along φ; the set of external identifiers remains the same, the correspondence between external identifiers and the new internal names is altered in the obvious way, and the models are translated along $\widehat{\varphi}$ (recall that $\widehat{\varphi}$ is the signature morphism from Σ to the new signature such that $\mathbf{Names}(\widehat{\varphi}) = \varphi$).

Of course, Σ does not have to be compatible along φ, which means that we are attempting to identify incompatible components (in the standard variant, this happens if n denotes a type and m denotes a value or if n and m denote values having different types). If this is the case, the elaboration of the elementary specification yields an error. Assumption 2 guarantees that otherwise the signature induced by Σ along φ exists.

We will not consider here the intricacies of sharing constraints for substructures inherited from Standard ML; a first approximation would be to regard such a sharing constraint as an abbreviation for a set of sharing constraints for the corresponding components of the substructures — see [ST 86] for the complete details.

The last way to extend EML signatures is to add a substructure of a given EML signature using the construct `structure` A : sig' where A is an atomic identifier and sig' is an EML signature, either defined here explicitly or taken from the environment. Intuitively, this adds sig' to the current EML signature, renaming each external identifier n of sig' to $A.n$. This is not enough, though, because it is necessary to avoid unintended sharing which may occur if sig' happens to use some of the same internal names as the current EML value. More formally, let $\langle N, \tau, \Sigma, C\rangle$ denote the current EML value and $\langle N', \tau', \Sigma', C'\rangle$ denote sig'. To elaborate `structure` A : sig' we have to choose a signature Σ'' isomorphic to Σ' (where i: $\Sigma' \to \Sigma''$ is the isomorphism) but having names disjoint from the names of Σ, modulo pervasive names. If such a Σ'' does not exist then we have (intuitively) run out of names which causes an error — but this cannot happen in any of the example institutions from section 3. Otherwise we take the union signature of Σ'' and Σ as the underlying signature of the result. It exists by assumptions 2 and 3. The class of models is the intersection of the classes of models of C and C' translated into the union signature (along $\iota_{\Sigma\subseteq\Sigma\cup\Sigma''}$ and $i;\iota_{\Sigma''\subseteq\Sigma\cup\Sigma''}$ respectively). The new external identifiers are the ones already present in N together with the identifier $A.n$ for each $n \in N'$ with the obvious association to internal names.

Axioms may be imposed on an EML signature to restrict its class of models. The syntactic representation of sentences from **INS** is used to present axioms using the construct `axiom` ax. To elaborate this in the current EML value $\langle N, \tau, \Sigma, C\rangle$ we first check if ax uses only the available external identifiers, i.e. if $ax \in \mathbf{Syn}(N)$. Then we translate it to syntax over the internal names $\mathbf{Names}(\Sigma)$, i.e. to $\mathbf{Syn}(\tau)(ax)$, and find the sentence θ in $\mathbf{Sen}(\Sigma)$ it represents, i.e. $\theta = repr_\Sigma(\mathbf{Syn}(\tau)(ax))$. This process may be compared to the process of parsing and typechecking in Standard ML. If it is unsuccessful, i.e. θ does not exist, an error occurs. Otherwise we restrict the class of models C to those which satisfy θ (according to the satisfaction relation $\models_\Sigma$).

The elaboration of elementary specifications as discussed above gives us the literal interpretation of all the axioms in an EML signature, giving an EML value $\langle N, \tau, \Sigma, C\rangle$. As discussed in section 2, we want to relax this interpretation using the notion of behavioural abstraction. This happens when we complete elaboration of any EML signature. To the class of models C we add all those Σ-models which are observationally equivalent to the models already in C with respect to a set Φ of observable sentences chosen to model an appropriate notion of behavioural equivalence (i.e. we add any model which satisfies exactly the same sentences of Φ as a model already in C). Naturally, the choice of Φ cannot be made in an institution-independent way. All we can provide to guide this choice is the set of names which should be viewed as having an externally fixed interpretation. These are the pervasive names together with those occurring in substructures, i.e. having non-atomic external identifiers. Thus we assume that when instantiating Extended ML in an institution with syntax we are provided with a function beh which assigns to any signature Σ and set of names $OBS \subseteq \mathbf{Names}(\Sigma)$ a set of Σ-sentences $beh(\Sigma, OBS)$ (more precisely, a set of open Σ-formulae — see [ST 85b]) such that (intuitively) observational equivalence with respect to $beh(\Sigma, OBS)$ models behavioural equivalence of Σ-models with respect to observable components OBS. For example, in the standard variant $beh(\Sigma, OBS)$ would yield the set of all equations between Σ-terms having sorts in OBS (and free variables of sorts in OBS).

4.2 EML structures

Like EML signatures, EML structures denote EML values $\langle N, \tau, \Sigma, C\rangle$; and like EML values denoting EML signatures, EML values denoting EML structures always contain the pervasive EML value as a full subvalue. However, an essential difference arises from the fact that EML structures can freely refer to components of previously-constructed EML structures. When an EML structure is constructed using pieces of other EML structures, sharing across structure boundaries may occur. This implies that EML structures cannot be treated as isolated entities; two components of different EML structures share iff they have the same internal name. We must ensure by our choice of internal names that such sharing does not arise unintentionally. The elaboration of structure expressions takes place in the context of an environment binding atomic identifiers to EML (structure) values so the internal names of all EML structures presently in existence are available.

As with EML signatures, the elaboration of an EML structure *str* starts by including the pervasive EML value and then proceeds by elaborating the elementary declarations in *str*. Elementary declarations in EML structures include all elementary specifications which can appear in EML signatures except for sharing constraints; sharing in EML structures arises by construction rather than by declaration. The semantics of these constructs in an EML structure context is more complicated than their interpretation in an EML signature context. To simplify slightly the description of the semantics below we will assume that identifiers may not be redeclared within a given structure. In order to permit redeclaration we would have to throw away components of the current EML value which are no longer accessible via external identifiers in appropriate places in the semantics — see [ST 86] for full details.

The first kind of elementary declaration extends the underlying signature without constraining the interpretation of the extension (in the standard variant this corresponds to declarations of new types/values without binding them, such as `type t` or `val v:t->t` but not `type t = int` or `val v =` *exp*). The syntax `extend along` $\iota\colon \Sigma \rightarrow \Sigma'$ and its semantics is the same as of the corresponding elementary specification in an EML signature. However, in order to avoid unintentional sharing we additionally have to ensure that the newly-introduced internal names are different from all those belonging to already-existing structures. We do this by changing the underlying signature of the result to an isomorphic one while preserving the names of the current signature Σ. This may be impossible if (intuitively) we have run out of names, in which case an error arises. This cannot happen in any of the example institutions from section 3.

The semantics of `new name` n `for` m is as before if m is an external identifier of the current EML value $\langle N, \tau, \Sigma, C\rangle$. If not, it must be of the form $A.p$ for some atomic identifier A naming an EML structure $\langle N', \tau', \Sigma', C'\rangle$ available in the environment where $p \in N'$. Then (intuitively) we have to grab the internal name and interpretation of p and add it to the current EML value under the external identifier n. More formally, let $\Sigma_p =_{def} \Sigma'|_{\{\tau'(p)\}}$ be the subsignature of Σ' generated by the internal name of p and let C_p be the restriction of the model class C' to Σ_p (along $\iota_{\Sigma_p \subseteq \Sigma'}$). We will need a similar operation of restricting an EML value to a subset of its external identifiers again in the sequel so let us define that for any EML value $\langle N', \tau', \Sigma', C'\rangle$ and set $X \subseteq N'$, the subvalue of $\langle N', \tau', \Sigma', C'\rangle$ generated by X is $\langle N', \tau', \Sigma', C'\rangle\big|_X =_{def} \langle X, \tau'\restriction X, \Sigma'|_{\tau'(X)}, C''\rangle$ where C'' is the class of $\Sigma'|_{\tau'(X)}$-models derived from C' using the signature inclusion $\iota_{\Sigma'|_{\tau'(X)} \subseteq \Sigma'}$.

Returning to the elaboration of the `new name` construct, the resulting EML value will have an underlying signature $\Sigma \cup \Sigma_p$, external identifiers $N \cup \{n\}$ with n assigned to the internal name $\tau'(p)$, and a class of models which is the intersection of the appropriate translations of C and C_p into the signature $\Sigma \cup \Sigma_p$. However, it may turn out that the union signature $\Sigma \cup \Sigma_p$ does not exist. Intuitively this may occur if Σ and Σ_p have incompatible structures on some common part. In order-sorted logic, suppose `A` is an EML structure containing sorts `t` and `t'` and consider the following example (where the constructs `require` and `val` are just convenient syntax for `extend along` in this particular institution):

```
structure B = struct structure C = A
                     require C.t < C.t'
                     val f : C.t -> C.t'
              end

structure D = struct structure E = A
                     require E.t' < E.t
                     new name g for B.f
              end
```

The subsignature of `B` generated by `f` must contain the sorts `A.t` $=$ `B.C.t` $=$ `D.E.t` and `A.t'` $=$ `B.C.t'` $=$ `D.E.t'` together with the subsort constraint `A.t`$\leq$`A.t'`. This is incompatible with the subsort constraint in `D` that `A.t'`$\leq$`A.t`. (Note that this does not contradict assumption 3; the common subsignature here is not full.) However, this kind of situation cannot occur in the other two institutions of section 3. We could have adjusted the category of order-sorted signatures to avoid this problem by making either of the following changes:

1. Require that all signature morphisms be full with respect to the ordering on sorts, i.e. for $\sigma\colon \Sigma \to \Sigma'$ and s, s' sorts in Σ, $\sigma(s) \leq \sigma(s')$ in Σ' implies $s \leq s'$ in Σ. This excludes signature inclusions which introduce new structure, as in the above example.

2. Allow the sorts to be preordered rather than ordered. This would imply that structure on signatures is never contradictory; in particular, the above example would succeed with both `A.t`$\leq$`A.t'` and `A.t'`$\leq$`A.t` as subsort constraints of `D` (implying the identity of the corresponding carriers).

One way to add a substructure to an EML structure is to declare it using the construct `structure` A : *sig* without constraining the interpretation of the substructure A further than the EML signature *sig* requires. The semantics of this is exactly the same as in an EML signature context. Note however that in order to avoid unintended sharing we have to pick new internal names which are different not only from the internal names of the current EML value but also from the internal names of all existing structures.

Another way to add a substructure is using the construct `structure` A `=` str' which introduces a new substructure A and binds it to the EML structure str'. If $\langle N, \tau, \Sigma, C\rangle$ is the current EML value and str' denotes the EML value $\langle N', \tau', \Sigma', C'\rangle$ then the underlying signature of the result is $\Sigma \cup \Sigma'$ (unless Σ and Σ' are incompatible, in which case an error is raised), the class of models is the intersection of the translations of C and C' to the signature $\Sigma \cup \Sigma'$, the external identifiers consist of the set N together with the name $A.p$ for each $p \in N'$, with the obvious association of internal names. Note that this is equivalent (modulo syntactic details) to a sequence of `new name` declarations of the form `new name` $A.n$ `for` n where n and its internal interpretation come from the EML value denoted by str. As a result the problem with incompatible signatures may arise here just as in the case of the `new name` construct.

The construct `structure` A : sig = str' also adds a substructure to the current structure. This is defined to be equivalent to `structure` A = $(str' : sig)$, where $str' : sig$ defines an EML structure as will be explained later.

As in EML signatures, it is possible to impose axioms which restrict the class of models of a structure using the construct `axiom` ax. However, in this case ax may make use of non-local identifiers and so its interpretation is more complicated. Let $\langle N, \tau, \Sigma, C\rangle$ be the current EML value and let U denote the set of all visible external identifiers, that is:

$$U =_{def} \{A.n \mid A \mapsto \langle N_A, \tau_A, \Sigma_A, C_A\rangle \text{ is in the environment and } n \in N_A\} \cup N$$

We have to check that $ax \in \mathbf{Syn}(U)$. If this is not the case (which happens when ax uses unavailable identifiers) then an error occurs; otherwise let U_{ax} be the set of external identifiers ax uses, i.e. the least subset of U such that $ax \in \mathbf{Syn}(U_{ax})$. This also determines the set of EML structures in the current environment ρ which are relevant for the interpretation of ax, where a structure A is relevant for ax if the set $N_A^{ax} =_{def} \{n \mid A.n \in U_{ax}\}$ is nonempty. Furthermore, consider the restriction of these EML structures to their minimal substructures necessary for the interpretation of ax: $\mathcal{STR}_{ax} =_{def} \{\rho(A)|_{N_A^{ax}} \mid N_A^{ax} \neq \emptyset\}$. Now we have to require that the set of underlying signatures of $\mathcal{STR}_{ax}$ is compatible and that Σ (the current signature) is compatible with it. If it is not, an error is raised as there can be no sensible interpretation for ax. Otherwise, we interpret ax in the union structure $\langle U_{ax}, \tau_{ax}, \Sigma_{ax}, C_{ax}\rangle$ where Σ_{ax} is the union of the underlying signatures of $\mathcal{STR}_{ax}$ and Σ, C_{ax} is the intersection of model classes of structures in $\mathcal{STR}_{ax}$ translated into Σ_{ax}, and τ_{ax} associates the external identifiers U_{ax} with the appropriate internal names (an external identifier of the form $A.n$ is mapped to the internal name corresponding to n in $\rho(A)$). We start this interpretation by finding the sentence $\theta =_{def} repr_{\Sigma_{ax}}(\mathbf{Syn}(\tau_{ax})(ax))$ which ax represents. If θ does not exist an error occurs. Finally, the resulting class of models is the restriction of the current model class C to those Σ models which may be derived from models in C_{ax} which additionally satisfy θ (according to $\models_{\Sigma_{ax}}$); the other components of the current EML value remain unchanged.

The intricacies of the above construction are mostly caused by the need to avoid forming a signature any larger than necessary to interpret ax. The problem is that otherwise the error of incompatibility may be raised even though ax has an interpretation in a smaller, compatible part of the environment. In the case of equational logic and categorical logic, compatibility of the underlying signatures of all existing EML structures is maintained (by our choice of internal names) and so a much simpler construction, using the union of all the structures in the environment to interpret ax, would suffice. This would also be the case if we had guaranteed that compatibility is preserved by the `extend along` construct. We have chosen instead to allow incompatible signatures to coexist as long as they do not interfere with each other.

Once an EML structure has been elaborated, it is possible to require it to fit a given EML signature using the construct $str : sig'$ which yields an error if str does not fit sig' and otherwise reduces str to the signature sig', forgetting all of its extraneous components. More formally, if str denotes $\langle N, \tau, \Sigma, C\rangle$ and sig' denotes $\langle N', \tau', \Sigma', C'\rangle$ then $str : sig'$ yields an error if $N' \not\subseteq N$ or if there is no signature morphism σ: $\Sigma' \to \Sigma$ such that for any $n' \in N'$, $\mathbf{Names}(\sigma)(\tau'(n')) = \tau(n')$.

$$
\begin{tikzcd}
N' \arrow[rr, phantom, "\subseteq"] \arrow[d, "\tau'"'] & & N \arrow[d, "\tau"] \\
\mathbf{Names}(\Sigma') \arrow[rr, "\mathbf{Names}(\sigma)"'] & & \mathbf{Names}(\Sigma) \\
\Sigma' \arrow[rr, dashed, "\sigma"] & & \Sigma
\end{tikzcd}
$$

Note if such a σ exists then it is unique since τ' is surjective (which comes from the fact that EML signatures are closed): there is at most one function from $\mathbf{Names}(\Sigma')$ to $\mathbf{Names}(\Sigma)$ such that the above diagram commutes and so (by faithfulness of $\mathbf{Names}$) there is at most one signature morphism σ with the required properties. Then we have to check that the models admitted by *str* satisfy the requirements of sig', i.e. we raise an error unless the Σ'-models derived from C using σ are included in C'. Finally, the resulting structure is $\langle N, \tau, \Sigma, C\rangle|_{N'}$. Note that the above description allows *str* to share more than is required by sig' and that this sharing is carried over to $str : sig'$.

Another way to construct a new EML structure out of a given one is to extract one of its substructures using the notation $str.A$. Let $\langle N, \tau, \Sigma, C\rangle$ be the EML value denoted by *str*. An error occurs if N contains no names of the form $A.n$. Otherwise the resulting EML value is $\langle N, \tau, \Sigma, C\rangle|_{\{A.n\in N\}}$ except that we change each external name $A.n$ to n.

A requirement we inherit from Standard ML is that once the construction of a structure is complete, it must be a closed EML value. This requirement is imposed for methodological reasons rather than because of technical necessity. It cannot be required at intermediate stages of the construction of a structure since the construct `new name` may add components to the underlying signature which remain temporarily anonymous. It is straightforward to check this condition since it is just the requirement that the assignment of internal names to external identifiers is surjective. But before checking for this it is necessary to throw away components of the underlying signature which were introduced during the construction of the structure but are no longer used. This is done by restricting the EML value to the set of all its external identifiers.

4.3 EML functors

Semantically, an EML functor acts as a function taking a list of actual parameters (which are EML structures over the formal parameter EML signatures) to a structure over the result EML signature. However, we have chosen to describe the semantics of functors using a macro-expansion mechanism and so no function of this kind appears explicitly in our semantics. The denotation of a functor

`functor` F`(`A_1`:`sig_1`,...,`A_n`:`sig_n `sharing` eq_1 `and ... and` eq_m`) :`sig `=` str

consists of a list of the formal parameter names $A_1, \ldots, A_n$, an EML value describing the combined formal parameter signatures, which is just the denotation of the EML signature

`sig structure` A_1`:`sig_1`; ...; structure` A_n`:`sig_n`;`
    `sharing` eq_1`; ...; sharing` eq_m
`end` ,

the body qualified by the result signature, $str : sig$, kept in its syntactic form, and the declaration-time environment. Applying this functor to a list of actual parameters $str_1, \ldots, str_n$ is done by elaborating the expression $str : sig$ in the declaration-time environment augmented by binding the parameter names $A_1, \ldots, A_n$ to the actual parameter values (after fitting them to the formal parameter signature).

5 Concluding remarks

In this paper we presented the institution-independent semantics of Extended ML (EML), a high-level specification language based on the ideas of Standard ML modules. This required extending the notion of an institution by adding facilities for manipulating the names associated with components of signatures and for syntactic representation of axioms. Such an extended institution we call an *institution with syntax*. This provided a framework sufficient to give a nearly complete institution-independent semantics of Extended ML. The only institution-dependent "leftovers" are:

1. the particular choice of the underlying pervasives which form a part of every EML signature and structure;

2. the set of observations required for the notion of behavioural equivalence of models; and

3. the part of the language for describing "elementary" signature extensions (i.e. those which are to be admitted in the given instantiation of EML).

The only other specification languages we know about which are defined in the framework of an arbitrary institution are Clear [BG 80] and ASL [ST 85c]. Extended ML as described here differs from them in at least the following respects:

1. Both Clear (in its institution-independent form!) and ASL provide nothing more than a bunch of specification-building operations. The syntax used to describe everything below the level of a specification and the meaning of that syntax is institution-dependent and therefore not supplied. In Extended ML we have attempted to deal even with the syntax of individual axioms; we have indicated, for example, where and how the problems of parsing and type-checking would appear.

2. The only explicit structuring facility Extended ML offers to its user is the notion of a *structure* (called a *functor* when parameterised) which is a direct extension of the notion of a structure in Standard ML. This brings the structure of specifications in Extended ML closer to the structure of programs and makes Extended ML more appropriate for *design* specification than either ASL or Clear (although see the comment at the end of 3 below), in that it allows the user not only to indicate the desired functional properties of a program/system, but also to design the structure of its implementation.

3. Extended ML is a wide spectrum language (see [Bau 81]), where programs are just specifications which happen to include only executable axioms. In [GB 80], Goguen and Burstall outline a scheme for developing programs from Clear specifications, but in this framework the specification language and programming language are kept separate although it is suggested that program modules could be put together using Clear's specification-building operations. Of course, both Clear and ASL could be used as wide spectrum languages; the difference is only that Extended ML was designed with this specific goal.

This work can be viewed from two different perspectives. From one point of view it is an exercise in applying the theory of institutions in the field of algebraic specifications. As in the case of e.g. [Tar 84,85], there is a limited amount which can be accomplished within the framework of an arbitrary institution as originally defined in [GB 84]; the "game" is to identify a minimal set of extra assumptions necessary for a particular purpose.

From another point of view, this work is a step towards a practical framework for formal program development. In [ST 85a] Extended ML was introduced as a vehicle for the development of Standard ML programs. However, the use of the standard algebraic framework there excluded (for example) the use of partial functions, polymorphic types and assignment. By providing an institution-independent semantics for Extended ML here, we make it possible in principle to remove these restrictions by plugging in an appropriate institution with syntax. For example, an institution permitting the use of partial functions based on [BW 82] is described in [ST 85c], and it is obvious how to extend this to an institution with syntax. Moreover, it is more or less apparent that Extended ML instantiated to this institution with syntax would allow one to treat partial functions in a satisfactory way. However, the situation with polymorphic types is much less clear. It is possible to construct an institution for polymorphism (see [SB 83] for some hints) and we do not anticipate problems in extending such an institution to an institution with syntax. But preliminary investigations along these lines indicate that Extended ML in this institution would not work as expected (the problem here has to do with the structure which is imposed on the set of type names in the presence of polymorphism). Assignment poses even bigger problems; at the moment we just do not know what an appropriate institution would be. Difficulties of this kind must be overcome before it will be possible to use Extended ML to develop programs in full Standard ML.

Another advantage of an institution-independent semantics for Extended ML is that it permits freedom in the choice of the logic used to specify Standard ML programs. Even more intriguing, it makes Extended ML ML-independent: since Extended ML uses only the modularisation facilities of Standard ML, it could be used to specify and develop programs in Prolog, Pascal, etc. by choosing an institution which includes program fragments in the desired language as sentences.

Acknowledgements

Our thanks to Rod Burstall and Joseph Goguen for their work on institutions, to Rod Burstall for the idea of adding axioms to a programming language, to David MacQueen for his work on modules for Standard ML and to Martin Wirsing for helping to develop ASL. This work was supported by the (U.K.) Science and Engineering Research Council, the Polish Academy of Sciences and the University of Edinburgh.

6 References

[Bar 74] Barwise, K.J. Axioms for abstract model theory. *Annals of Math. Logic 7* pp. 221-265.

[Bau 81] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development. Report TUM-I8104, Technische Univ. München. See also: *The Wide Spectrum Language CIP-L.* Springer LNCS 183 (1985).

[BW 82] Broy, M. and Wirsing, M. Partial abstract types. *Acta Informatica 18* pp. 47-64.

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of Clear, a specification language. *Proc. of Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, pp. 292-332.

[Ehr 79] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. Report 82, Univ. of Dortmund. Also in: *Journal of the Assoc. for Computing Machinery 29* pp. 206-227 (1982).

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. *Theoretical Computer Science 20* pp. 209-263.

[GB 80] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, SRI International.

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop* (E. Clarke and D. Kozen, eds.), Carnegie-Mellon University. Springer LNCS 164, pp. 221-256.

[GB 86] Goguen, J.A. and Burstall, R.M. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. *Proc. Workshop on Category Theory and Computer Programming*, Guildford (this volume). Springer LNCS.

[GJM 85] Goguen, J.A., Jouannaud, J.-P. and Meseguer, J. Operational semantics for order-sorted algebra. *Proc. 12th Intl. Colloq. on Automata, Languages and Programming*, Nafplion, Greece. Springer LNCS 194, pp. 221-231.

[GTW 76] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM research report RC 6487. Also in: Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149 (1978).

[Gut 75] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto.

[LHKO 84] Luckham, D.C., von Henke, F.W., Krieg-Brückner, B. and Owe, O. Anna: a language for annotating Ada programs (preliminary reference manual). Technical report 84-248, Computer Systems Laboratory, Stanford University.

[MacL 71] MacLane, S. *Categories for the Working Mathematician.* Springer.

[MacQ 85] MacQueen, D.B. Modules for Standard ML. *Polymorphism 2*, 2. See also: *Proc. 1984 ACM Symp. on LISP and Functional Programming*, Austin, Texas, pp. 198-207.

[Mil 85] Milner, R.G. The Standard ML core language. *Polymorphism 2*, 2. See also: A proposal for Standard ML. *Proc. 1984 ACM Symp. on LISP and Functional Programming*, Austin, Texas, pp. 184-197.

[NY 83] Nakajima, R. and Yuasa, T. (eds.) *The IOTA Programming System: A Modular Programming Environment.* Springer LNCS 160.

[Rei 84] Reichel, H. Behavioural validity of conditional equations in abstract data types. *Contributions to General Algebra 3: Proc. of the Vienna Conference.* Verlag Hölder-Pichler-Tempsky, pp. 301-324.

[SB 83] Sannella, D.T. and Burstall, R.M. Structured theories in LCF. *Proc. 8th Colloq. on Trees in Algebra and Programming*, L'Aquila, Italy. Springer LNCS 159, pp. 377-391.

[ST 85a] Sannella, D.T. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67-77.

[ST 85b] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. Report CSR-172-84, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Journal of Computer and Systems Sciences.* Extended abstract in: *Proc. 10th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Berlin. Springer LNCS 185, pp. 308-322.

[ST 85c] Sannella, D.T. and Tarlecki, A. Specifications in an arbitrary institution. Report CSR-184-85, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Information and Control.* See also: Building specifications in an arbitrary institution, *Proc. Intl. Symposium on Semantics of Data Types*, Sophia-Antipolis. Springer LNCS 173, pp. 337-356 (1984).

[ST 86] Sannella, D.T. and Tarlecki, A. An institution-independent semantics for Extended ML. Research report, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (in preparation).

[SW 83] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh. Extended abstract in: *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, pp. 413-427.

[Tar 84] Tarlecki, A. Quasi-varieties in abstract algebraic institutions. Report CSR-173-84, Dept. of Computer Science, Univ. of Edinburgh; to appear in *Journal of Computer and Systems Sciences.*

[Tar 85] Tarlecki, A. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science 37* pp. 269-304.

[Wir 83] Wirsing, M. Structured algebraic specifications: a kernel language. Habilitation thesis, Technische Univ. München.

[Zil 74] Zilles, S.N. Algebraic specification of data types. Computation Structures Group memo 119, Laboratory for Computer Science, MIT.