Toward formal development of ML programs: foundations and methodology*

— Preliminary version† —

Donald Sannella
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

Andrzej Tarlecki
Institute of Computer Science
Polish Academy of Sciences
Warsaw

Abstract

A formal methodology is presented for the systematic evolution of modular Standard ML programs from specifications by means of verified refinement steps, in the framework of the Extended ML specification language. Program development proceeds via a sequence of *design* (modular decomposition), *coding* and *refinement* steps. For each of these three kinds of steps, conditions are given which ensure the correctness of the result. These conditions seem to be as weak as possible under the constraint of being expressible as "local" interface matching requirements. Interfaces are only required to match up to behavioural equivalence, which is seen as vital to the use of data abstraction in program development.

*An extended abstract of this paper will appear in *Proc. Colloq. on Current Issues in Programming Languages*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Barcelona, Springer LNCS (1989).

†A later version will take into account the recent changes to ML described in [HMT 88]. The relevant changes concern mainly functors with multiple arguments.

Contents

1 Introduction

The ultimate goal of work on algebraic specification is to provide a formal basis for program development to support a methodology for the systematic evolution of programs from specifications by means of verified refinement steps.

In this paper we present such a methodology aimed at the development of programs in the Standard ML programming language [HMM 86], [Har 86]. We are interested particularly in the semantic and foundational underpinnings of modular program development and in formulating precise conditions which ensure the correctness of refinement steps, rather than in informal rules governing good programming practice. We build on previous work on the foundations of algebraic specifications, on behavioural equivalence, on notions of specification refinement, on the wide-spectrum programming/specification language Extended ML, and on data abstraction in modular programming. In this introduction we will briefly review these topics and then give an overview of the methodology as presented in this paper. We assume that the reader is familiar with the basic algebraic notions in e.g. [GTW 78] (cf. [BG 82]). In this introduction we also assume a passing acquaintance with the terminology of Standard ML, to be introduced in Section 2.

Algebraic specifications

The most fundamental assumption in work on algebraic specification is the view of software systems as algebras, abstracting away from details of algorithms and code and focussing on data representation and functional behaviour of programs. A specification is a document describing some class of algebras, defining in this indirect way which programs are acceptable as realisations. So whatever specification formalism we use, we assume that any specification SP determines an algebraic signature $Sig[SP]$ and a class of algebras $Mod[SP]$ over this signature, called the *models* of SP. No further assumptions are needed for most purposes. We view the use of equational logic and initial algebra semantics (as in [GTW 78] and [EM 85]) as just one possible choice which happens to be very convenient for certain purposes, e.g. rapid prototyping of specifications via term rewriting. See [SWi 83], [ST 85a] and [ST 88a] for details of this point of view.

Behavioural equivalence

It may be argued that a software system should be accepted as a realisation of a specification SP as long as it "behaves like" a model of SP even if it does not satisfy SP exactly. This intuition may be made precise by introducing an appropriate notion of *behavioural equivalence* of algebras. Then the interpretation of SP may be relaxed modulo this behavioural equivalence. Various notions of behavioural equivalence have been studied in [GGM 76], [Rei 81], [GM 82], [ST 87], [NO 88] and elsewhere; the idea goes back at least to work on automata theory in the 1950's [Moo 56].

Specification refinement

A theory of formal program development by stepwise refinement of specifications requires a precise definition of the notion of refinement and when a refinement step is considered to be correct. In the following paragraph we summarize the work on this topic presented in [ST 88b]; other relevant papers include [GB 80], [Ehr 82], [EKMP 82], [GM 82], [Wand 82], [Gan 83], [Lip 83], [Ore 83] and many others.

Intuitively, refining a specification corresponds to making design decisions, thus restricting the class of acceptable models. The simplest notion of refinement of one specification $SP1$ to another

$SP2$ would only require inclusion of model classes, i.e. $Mod[SP2] \subseteq Mod[SP1]$. A more realistic view involves a construction κ taking models of $SP2$ to models of $SP1$; we write $SP1 \underset{\kappa}{\rightsquigarrow} SP2$. Here, a construction is just a function $\kappa : Alg(Sig[SP2]) \rightarrow Alg(Sig[SP1])$ between classes of algebras; examples include forgetting types/values and free extension (subsuming extension of an algebra by "code"). Using these so-called *constructor implementations*, the program development process consists of a sequence of consecutive implementation steps:

$$SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n$$

where SP_0 is the original high-level specification of requirements. Then, the composition of constructions $\kappa_n;\cdots;\kappa_2;\kappa_1$ forms a "parameterised program" (cf. [Gog 84]) which implements SP_0 in terms of SP_n.[1] This assumes that the class of constructions is closed under composition. If SP_n is a specification for which a realisation A_n is already available, then the application of this composed construction to A_n yields a realisation of SP_0.

Extended ML

We have proposed the specification language Extended ML [ST 85b], [ST 86] as a vehicle for formal development of programs in the programming language Standard ML. Extended ML is based on the modularisation facilities for Standard ML proposed in [MacQ 86], which are designed to allow large Standard ML programs to be structured into modules with explicitly-specified interfaces. In Extended ML these are enhanced by allowing more information in module interfaces (axioms in ML *signatures*) and less information in module bodies (axioms in place of code in ML *structures* and *functors*). Standard ML forms a subset of Extended ML, since Standard ML datatype and function definitions are just axioms of a certain special form. Thus Extended ML is a *wide-spectrum* language in the spirit of CIP-L [Bau 85]. The semantics of Extended ML is defined in terms of the primitive specification-building operations of the ASL kernel specification language [SWi 83], [ST 88a].

Data abstraction in modular programming

A general theory of modular program development using data abstraction, incorporating ideas going back to [Hoa 72] and [Par 72], is presented in [Sch 86]. The main issue, referred to as "the correctness problem of data abstraction", is why it is possible for the implementor of a specification to provide a realisation which is correct only up to behavioural equivalence, while users of the result may view it as if it satisfied the specification exactly. A very rough explanation of this apparent paradox is that users are not able to take advantage of the properties which distinguish "exact" models of a specification from their behavioural approximations. It is argued that this property, called *stability*, should be required of any programming language designed to support data abstraction.

A technical framework to deal with this phenomenon is developed in [Sch 86] which copes not just with behavioural equivalence but more generally with any so-called "representation relation". In this paper we borrow many of the concepts and results formulated there, applying them in our context where we deal with behavioural equivalence only.

The central observation which led us to the ideas presented in the current paper was that Standard ML functors may be used to code constructions in the above sense. Additionally, Extended ML

[1] Please note that semicolon denotes function composition here, *not* sequential composition of commands. Also, $\kappa_1, \ldots, \kappa_n$ are functions which operate on algebras, *not* on data values.

allows us to specify such constructions before they are actually coded. ML's modularisation facilities are designed to guarantee their composability by analogy with function composition. This gives rise to a view of program development which is more complex but also methodologically more appealing than the one presented in [ST 88b].

A programming task is presented as an Extended ML functor heading, i.e. an Extended ML signature SP_n specifying structures to which the functor may be applied, and an Extended ML signature SP_0 specifying the required result structure. Recall that Extended ML signatures may contain axioms. Rather than proceeding from SP_0 to SP_1, and then from SP_1 to SP_2, ..., and then from SP_{n-1} to SP_n as described above, we take a more global view with development steps of the following kinds:

Design step: Sketch the implementation process $SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n$ without coding the constructions $\kappa_1, \ldots, \kappa_n$. This gives rise to specifications of functors $\kappa_1, \ldots, \kappa_n$ which are then viewed as separate programming tasks in their own right to which the same methodology applies. The composition of these functors results in a construction which implements SP_0 in terms of SP_n. The design may have a more complex structure than this linear notation suggests, since functors may have multiple arguments and the same functor may be used in different places.

Coding step: Code a construction by providing a functor body in the form of an encapsulated structure containing type and value definitions. It is also possible to use an "abstract program" here, i.e. an Extended ML functor body containing axioms.

Refinement step: Further refine abstract programs in a stepwise fashion by providing successively more concrete (but possibly still non-executable) versions which fill in some of the decisions left open by the more abstract version.

The paper is organized as follows. Section 2 gives an overview of the modularisation facilities of the Standard ML programming language and reviews the main features of and motivations behind the Extended ML specification language. This is mainly included in order to make this paper self-contained. Section 3 recalls the notion of behavioural equivalence and introduces the new notion of *behavioural consequence* which plays a basic role in verification conditions ensuring the correctness of development steps. Some preliminary results are given for proving behavioural consequence between loose specifications; as far as we know this topic has not been directly addressed in the literature. Section 4 presents the semantics of Extended ML functors; this is different from the previous version presented in [ST 85b]. The concept of *universal correctness* of an Extended ML functor with respect to its interface specifications is introduced following [Sch 86]. A functor is universally correct if it produces a result which satisfies the output interface up to behavioural equivalence whenever it is given an argument satisfying the input interface up to behavioural equivalence.

Sections 5 and 6 present the methodology of program development. Section 5 discusses design steps in which a functor is defined by decomposition into a collection of simpler functors. Three simple but representative special situations are studied and verification conditions ensuring the correctness of the decomposition are formulated and proved sound. The general case is also discussed. The longest proofs from this section are left to the appendices. Section 6 is about coding and refinement steps. Following [Sch 86], we present universal correctness as the conjunction of three properties: *simple correctness*, *simple consistency* and *stability*. A functor is simply correct if it produces a result which satisfies the output interface up to behavioural equivalence whenever it is given an argument which *exactly* satisfies the input interface (recall that for universal correctness, arguments which only satisfy the input interface up to behavioural equivalence must also be considered). A further difference is that universal correctness takes account of the global environment in which the functor is used. As suggested above, stability is assumed to be ensured for Standard ML functors since Standard ML is designed to support data abstraction, and simple consistency holds for any program. We formulate verification conditions which guarantee simple correctness of directly coded functors and functors produced by successive refinement steps. Thus, once a final Standard ML functor is obtained it will be simply correct, simply consistent and stable, and hence universally correct.

Section 7 presents an example of the application of this methodology. It is not intended to display all of the most subtle points discussed in the paper, but rather to demonstrate how a software system may be developed by means of a series of mostly very routine steps. Section 8 contains some conclusions and discusses areas for further research.

2 An overview of Extended ML

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 85b]. The version of Extended ML used in this paper is different in certain details from the one presented in [ST 85b] and [ST 86] but the general motivation and ideas and the overall appearance of specifications remains the same. The changes which have been introduced were motivated by the methodological issues to be discussed in this paper. We indicate the specific points of difference in this section; a revised formal semantics will be given in [ST 89].

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. It will be sufficient to know that a sequence 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 86], and a formal semantics is in [HMT 87].[2]

Extended ML is based on the modularisation facilities for Standard ML proposed in [MacQ 86]. 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*) 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*[3] are "parameterised" structures; the application of a functor to a structure yields a structure. 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 and/or values[4]) in the hierarchy are identical or *shared*. This issue will be discussed later in this section.

[2] A new version of this semantics [HMT 88] incorporates changes to ML which have not been taken into account here. The main changes of relevance here concern functors with multiple arguments.

[3] Functors were called *modules* in an early version of [MacQ 86] and in [ST 85b]. Category theorists should note that ML functors have no morphism part, and that ML supports no explicit notion of morphism between structures.

[4] Standard ML does not support sharing declarations for values. Extended ML supports this on the grounds that it

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
       datatype sequence = empty | cons of Elements.elem * sequence
       val sort : sequence -> sequence
   end

functor Sort(PO : POSig) : SortSig =
   struct structure Elements = PO
          datatype sequence = empty | cons of Elements.elem * sequence
          fun insert(a,empty) = cons(a,empty)
            | insert(a,cons(b,s)) = if Elements.le(a,b)
                                    then cons(a,cons(b,s))
                                    else cons(b,insert(a,s))
          fun sort empty = empty
            | sort(cons(a,s)) = insert(a,sort s)
   end

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

structure SortInt = Sort(IntPO)
```

Now, `SortInt.sort` may be applied to the sequence

$$\texttt{SortInt.cons}(11, \texttt{SortInt.cons}(5, \texttt{SortInt.cons}(8, \texttt{SortInt.empty})))$$

to yield

$$\texttt{SortInt.cons}(5, \texttt{SortInt.cons}(8, \texttt{SortInt.cons}(11, \texttt{SortInt.empty}))).$$

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 `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 type `int` is used together with the pervasive value `<=` (i.e. $\leq$). The pervasive types and values may be regarded as forming a structure `Perv` 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

---

is easier and more uniform to treat types and values in the same way.

the name n rather than the name Perv.n). The declaration `datatype sequence = ...` defines a new type `sequence` having as values all terms built using the constant `empty : sequence` and the function `cons : Elements.elem * sequence -> sequence` (`empty` and `cons` are called *constructors*). This allows us to define `insert` and `sort` by cases using `empty` and `cons` for case selection and variable binding.

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 generally provide enough information to permit proving program correctness (for example). To make signatures more useful as interfaces of structures in program specification and development, we 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 le(x,x)
       axiom le(x,y) & le(y,x) => x=y
       axiom le(x,y) & le(y,z) => le(x,z)
   end
```

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

Axioms are expressions of type `bool`. Using such an expression as an axiom amounts to an assertion that the value of the expression is `true` for all values of its free variables. Axioms may be built using functions such as `&`, `=>` and `<=>` and quantifiers such as `exists` and `forall` (with the usual precedences of these symbols), and the function `=` may be used to compare values of any type. This is equivalent to using first-order equational logic. Of course, Standard ML code will not contain quantifiers or use `=` except on types which admit equality according to Standard ML.

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, we allow structures to contain a mixture of ML code and non-executable axioms. Functors can 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:

```
functor Sort(PO : POSig) : SortSig sharing Elements=PO =
   struct structure Elements = PO
          datatype sequence = empty | cons of Elements.elem * sequence
          fun append(empty,s) = s
            | append(cons(a,s1),s2) = cons(a,append(s1,s2))
```

[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). We will use the term *algebraic signature* to refer to ordinary many-sorted signatures.

```
        fun member(a,empty) = false
          | member(a,cons(b,s)) = if le(a,b) andalso le(b,a)
                                  then true else member(a,s)
        val insert : Elements.elem * sequence -> sequence
        axiom member(a,insert(a,s))
        axiom insert(a,s) = append(s1,(cons(a,s2)))
                     => append(s1,s2) = s
                        & (member(a1,s1) => Elements.le(a1,a))
                        & (member(a2,s2) => Elements.le(a,a2))
        fun sort empty = empty
          | sort(cons(a,s)) = insert(a,sort s)
end
```

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. The sharing constraint `sharing Elements=PO` in the functor heading asserts that the substructure `Elements` of the structure built by the functor is identical to the actual parameter structure.

In Standard ML and in the version of Extended ML described in [ST 85b] and [ST 86], the interface of a functor is taken to be the signatures in the heading augmented by inferred sharing (sharing *by construction* in [MacQ 86]). For methodological reasons which will be clarified in later sections, we view the interface of a functor as containing no more information than is explicitly given in the functor heading. (This means that functors in Extended ML are actually parameterised *abstractions* in the sense of [MacQ 86].) Sharing constraints of the kind used in the heading of `Sort` (which actually play the role of sharing *declarations* here) help to make this regime work in practice.

Standard ML allows signatures to include sharing constraints which refer to the external structure environment [HMT 87]. We will assume that Extended ML signatures used as functor result specifications do not include such references to external structures. This assumption is purely for simplicity of presentation; our methodology (including all correctness results) can be extended to handle this case. Signatures with such external references are not really necessary anyway since any Standard ML system of functors may be transformed to the form we require by including the external structures in question as explicit functor parameters.

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 apart from the new kind of sharing declaration discussed above 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). We also allow signatures to contain hidden types and values which sometimes must be added to specify other types and values. We draw a box around hidden types and values (and the axioms which specify them) as syntactic sugar for ML's local declaration construct. Signatures and structures both denote classes of algebras.[6] To be more exact, each signature or structure determines a many-sorted algebraic signature where sorts are type names and operation names are value names and the typing of values determines the rank of operation names. Because of type definitions like `type t = s` in structures and sharing constraints in signatures, in general there may be several names for a single type or value.

[6]The standard notion of algebra is not sufficient to handle features of Standard ML such as polymorphism, higher-order functions or exceptions — see comments at the end of this section on this point.

We cope with this by assuming that the names which occur in the algebraic signature associated with a structure or signature are unique *internal* semantic-level names which are associated with one or more *external* identifiers which may appear in Extended ML text. Two types or values share iff they have the same internal identifier. A structure or signature determines the class of algebras over its associated (internal) algebraic signature which satisfy its axioms; recall that code in structures is just a sequence of axioms of a certain special form.

The role of signatures as interfaces suggests that they should be regarded only as descriptions of the externally observable behaviour of structures. This amounts to not distinguishing between *behaviourally equivalent* algebras in which computations produce the same results of "external" types. (See [ST 87] for more motivation for the use of this notion here and for much more technical detail.) In the version of Extended ML in [ST 85b] and [ST 86] this led us to define the semantics of signatures by first obtaining the class of algebras which "literally" satisfy the axioms and then *behaviourally abstracting* (closing under behavioural equivalence with respect to a certain fixed subset of the types in the signature) to obtain the class of algebras which "behaviourally" satisfy the axioms (cf. [Rei 84]). In the current version of Extended ML we use different technicalities to implement these ideas. The semantics of signatures does not include the behavioural abstraction step; axioms in signatures are treated literally, just as in structures. When a signature is used as an interface, behavioural abstraction is invoked to relax its interpretation. The advantage of this treatment is that the types which are to be regarded as external depend on the context in which the signature is used. This extra flexibility turns out to be crucial for the methodology we develop in this paper. See the sequel for details.

As was outlined in [ST 86], Extended ML is actually entirely independent of Standard ML (although *not* of Standard ML's modularisation facilities, which we regard as separate from Standard ML itself). This is due to the fact that the semantics of Extended ML in [ST 86] was parameterised by an arbitrary *institution* [GB 84] which means that we are free to adopt any logical system for writing specifications. (More precisely, we can select any notion of algebraic signature, algebra and axiom and any definition of the satisfaction of an axiom by an algebra, provided that a few simple consistency conditions hold.) This not only allows us to use any desired specification style (taking equations, first-order formulae or maybe Horn clauses as axioms and taking ordinary many-sorted algebras, continuous algebras or perhaps polymorphic error algebras as algebras) but also to adopt any programming language with an algebraic-style formal definition for writing code. We are not going to follow this line in this paper: we present our ideas in the framework of total many-sorted algebras with first-order equational formulae as axioms as above, using a purely functional subset of Standard ML without polymorphism or higher-order functions for writing code. This is mainly to take advantage of the reader's intuition and to simplify some technicalities. We discuss in the conclusion how the concepts we develop may be generalized to an arbitrary institution.

The above paragraphs sketched some of the main ideas behind the formal semantics of Extended ML [ST 86], [ST 89]. The detailed treatment of the external/internal identifier distinction and sharing, a consequence of Standard ML's naming conventions, makes the semantics a little involved. In this paper we will not belabour this distinction: external names will be implicitly identified with their corresponding internal names when convenient. Because internal names are used to keep track of sharing, this means that sharing is implicitly taken into account as appropriate. The semantics is defined entirely in terms of the ASL kernel specification language [SWi 83], [Wir 86], [ST 88a]. This means that work in the context of ASL on implementation or refinement of specifications [ST 88b], observational and behavioural equivalence [ST 87] and proving theorems in specifications [ST 88a] provides a rich theoretical background for the methodology we present here.

3 Behavioural equivalence

In the previous section we mentioned the notion of behavioural equivalence in connection with program interfaces (Extended ML signatures). Intuitively, we don't care exactly how a program works if we are going to use it as a component in a larger system; we only care about the behaviour the program exhibits, where the behaviour is determined just by the answers which are obtained from computations the program may perform. We say (informally) that two Σ-algebras are *behaviourally equivalent* with respect to a set *OBS* of *observable sorts* if it is not possible to distinguish between them by evaluating Σ-terms which produce a result of observable sort. For example, suppose Σ contains the sorts *nat*, *bool* and *bunch* and the operations $empty\colon \rightarrow bunch$, $add\colon nat, bunch \rightarrow bunch$ and $\in\colon nat, bunch \rightarrow bool$ (as well as the usual operations on *nat* and *bool*), and suppose A and B are Σ-algebras with

$$
\begin{aligned}
|A_{bunch}| &= \text{the set of finite sets of natural numbers} \\
|B_{bunch}| &= \text{the set of finite lists of natural numbers}
\end{aligned}
$$

with the operations and the remaining carriers defined in the obvious way (but B does *not* contain operations like *cons*, *car* and *cdr*). Then A and B are behaviourally equivalent with respect to $\{bool\}$ since every term of sort *bool* has the same value in both algebras (the interesting terms are of the form $m \in add(a_1, \ldots, add(a_n, empty)\ldots))$. Note that A and B are not isomorphic.

The idea of behavioural equivalence may be formalized as follows.

Definition 3.1 Let Σ be a many-sorted algebraic signature with a distinguished set $OBS \subseteq sorts(\Sigma)$ of *observable* sorts. Suppose A, B are Σ-algebras with $|A|_s = |B|_s$ for all $s \in OBS$. A and B are *behaviourally equivalent with respect to OBS*, written $A \equiv_{OBS} B$, if for any term t of a sort in *OBS* containing only variables X of sorts in *OBS* and any valuation $v\colon X \rightarrow |A|_{OBS}$ $(= |B|_{OBS})$, $t_A(v) = t_B(v)$ (we use the notation $t_A(v)$ for the value of t in A under v).

There is a model-theoretic formulation of this definition due to [Sch 86] (Theorem 4.4.6, p. 244):

Lemma 3.2 *Given an algebraic signature Σ with a distinguished set $OBS \subseteq sorts(\Sigma)$ and Σ-algebras A and B, $A \equiv_{OBS} B$ iff there exists a $sorts(\Sigma)$-sorted relation $R = \langle R_s \subseteq |A|_s \times |B|_s\rangle_{s\in sorts(\Sigma)}$ which is the identity on sorts in OBS and which satisfies the usual congruence property: for any $f\colon s_1 \times \cdots \times s_n \rightarrow s$ in Σ, if $\langle a_1, b_1\rangle \in R_{s_1}, \ldots, \langle a_n, b_n\rangle \in R_{s_n}$ then $\langle f_A(a_1, \ldots, a_n), f_B(b_1, \ldots, b_n)\rangle \in R_s$.*
$\Box$

This model-theoretic criterion is useful for proving that two specific algebras are behaviourally equivalent. However, in formal program development we are rarely faced with this problem. Rather, we want to know that a certain loose specification (which may have many non-isomorphic models) matches another loose specification up to behavioural equivalence. That is, given two loose specifications *SP*1 and *SP*2 over the same signature and a distinguished set *OBS* of observable sorts, we want to prove that *SP*2 is a behavioural consequence of *SP*1 with respect to *OBS* in the following sense:

Definition 3.3 Let Σ be an algebraic signature with a distinguished set of observable sorts $OBS \subseteq sorts(\Sigma)$. Let *SP*1 and *SP*2 be specifications over Σ, let A be a Σ-algebra, and let K be a class of Σ-algebras.

- *A satisfies $SP2$ up to behavioural equivalence with respect to OBS*, written $A \models^{OBS} SP2$, if there exists an algebra $B \in Mod[SP2]$ such that $A \equiv_{OBS} B$.
- *K satisfies $SP2$ up to behavioural equivalence with respect to OBS*, written $K \models^{OBS} SP2$, if every algebra in K satisfies $SP2$ up to behavioural equivalence w.r.t. OBS.
- $SP2$ is a *behavioural consequence of SP1 with respect to OBS*, written $SP1 \models^{OBS} SP2$, if $Mod[SP1] \models^{OBS} SP2$.

A typical situation which involves proving behavioural consequence is checking whether an Extended ML structure fits an Extended ML signature. Since the signature is viewed as an interface defining the externally observable behaviour of the structure, we do not require that the structure satisfies the axioms in the signature literally, but only up to behavioural equivalence with respect to an appropriate set of observable sorts. For top-level structures the sorts corresponding to pervasive types are taken as observable. For structures occurring inside functor bodies, it is appropriate to take additionally some sorts in the functor parameters as observable. In both cases, we require the signature (which is a specification) to be a behavioural consequence of the structure (which is a specification as well), except that we permit the algebraic signature associated with the structure to be "larger" than the one associated with the signature; more on this point later.

As far as we know, the important problem of proving that one specification is a behavioural consequence of another has not been addressed directly in the literature although of course the "pointwise" characterization of behavioural equivalence given in Lemma 3.2 may be used in proving facts of this kind and some related material may be found in [Gan 83], [ST 87] and [NO 88]. The work of Reichel [Rei 84] on a logic for behavioural validity seems relevant here as well. The following results address this problem by giving proof-theoretic sufficient conditions for behavioural equivalence. More work needs to be done here but the theorems below cover the most obvious cases including those which are normally considered in work on algebraic specification. These results may be viewed as reformulations of known connections between behavioural equivalence, terminal models, and characterisations of formulae which hold in the terminal model of a specification.

Let us consider two specifications $SP1$ and $SP2$ over the same algebraic signature Σ (i.e. $Sig[SP1] = Sig[SP2] = \Sigma$) and a set $OBS \subseteq sorts(\Sigma)$ of observable sorts. First, notice that behavioural consequence is weaker than ordinary consequence. Although trivial, this result treats the most common case and so it is worth stating:

Proposition 3.4 *If $SP1 \models SP2$ then $SP1 \models^{OBS} SP2$.* □

In order to formulate further results we have to recall some standard notation and terminology.

For any $sorts(\Sigma)$-sorted set X of variables, $T_\Sigma(X)$ denotes the $sorts(\Sigma)$-sorted set of Σ-terms with variables X. If x is a variable of a sort $s \in sorts(\Sigma)$ such that $x \notin X$ then $T_\Sigma(X \cup \{x{:}s\})$ is the set of *contexts* for sort s. For any context $\Gamma \in T_\Sigma(X \cup \{x{:}s\})$ and term $t \in T_\Sigma(X)_s$, $\Gamma(t)$ denotes the term resulting from Γ by substituting t for all occurrences of x. A *substitution* (of terms with variables Y for variables X) is a $sorts(\Sigma)$-sorted map $\theta : X \to T_\Sigma(Y)$, where Y is a $sorts(\Sigma)$-sorted set of variables. For any term $t \in T_\Sigma(X)$, $t[\theta] \in T_\Sigma(Y)$ denotes the term with variables Y resulting from t by substituting $\theta(x)$ for each occurrence of every variable $x \in X$.

A *conditional Σ-equation with variables X* is a closed formula of the form

$$\forall X.\ (\bigwedge_{i \in I} t_i = t'_i) \Rightarrow t = t'$$

where $t, t' \in T_\Sigma(X)_s$ for some $s \in sorts(\Sigma)$, I is an arbitrary set of indices, and for $i \in I$, $t_i, t'_i \in T_\Sigma(X)_{s_i}$ for some $s_i \in sorts(\Sigma)$. We say that a conditional equation of the above form *has premises of observable sorts* if for all $i \in I$, s_i is observable (i.e. $s_i \in OBS$); we say that it is *observable* if it has premises of observable sorts, the conclusion is of an observable sort (i.e. $s \in OBS$) and all variables are of observable sorts (i.e. $X_r = \emptyset$ for $r \notin OBS$).

For any conditional equation φ of the above form with premises of observable sorts, *observable consequences of* φ are defined "syntactically" as observable conditional equations of the form

$$\forall Y.\ (\bigwedge_{i \in I} t_i[\theta] = t'_i[\theta]) \Rightarrow \Gamma(t[\theta]) = \Gamma(t'[\theta])$$

where Y is a set of variables of observable sorts (i.e. $Y_r = \emptyset$ for $r \notin OBS$), $\Gamma \in T_\Sigma(Y \cup \{x{:}s\})_{s'}$ is a context of an observable sort $s' \in OBS$ for the sort s of t and t', and $\theta : X \to T_\Sigma(Y)$ is a substitution. The set of all observable consequences of φ will be denoted by $ObsCon(\varphi)$. For any set Φ of conditional equations with observable premises, the set $ObsCon(\Phi)$ of observable consequences of Φ is defined "pointwise", i.e. $ObsCon(\Phi) = \bigcup_{\varphi \in \Phi} ObsCon(\varphi)$.

Now, the idea is that a Σ-algebra A satisfies a set of conditional equations with observable premises up to behavioural equivalence if and only if it satisfies its observable consequences in the usual sense. This is the essence of the following theorem:

Theorem 3.5 *Consider two specifications $SP1$ and $SP2$ over the same algebraic signature Σ and a set $OBS \subseteq sorts(\Sigma)$ of observable sorts. Suppose that $SP2$ is given as a set Φ of conditional Σ-equations with observable premises. Then $SP1 \models^{OBS} SP2$ iff $SP1 \models ObsCon(\Phi)$.*

Proof For the "only if" part, note that the observable consequences of a conditional equation are indeed consequences of it in the usual sense and, moreover, observable consequences of a conditional equation with observable premises are "observable" (if two Σ-algebras are behaviourally equivalent w.r.t. OBS then they satisfy exactly the same observable conditional equations).

For the "if" part, consider an arbitrary model $A \in Mod[SP1]$. From the assumption we have $A \models ObsCon(\Phi)$. We have to construct a Σ-algebra Z which is a model of $SP2$, i.e. $Z \models \Phi$, and which is behaviourally equivalent to A.

Consider the class of Σ-algebras which are generated by their carriers of observable sorts and which are behaviourally equivalent to A. It is well-known that this class contains a terminal algebra Z (cf. e.g. [BPW 84]). Z may be constructed as follows:

1. Consider the subalgebra $\langle A \rangle_{OBS}$ of A generated by $|A|_{OBS}$. The carriers of $\langle A \rangle_{OBS}$ may be defined as follows:

$$|\langle A \rangle_{OBS}|_s = \{t_A(v) \mid t \in T_\Sigma(Y)_s, Y \text{ is a set of variables of observable sorts}, v : Y \to |A|_{OBS}\}.$$

2. Define the Nerode congruence on $\langle A \rangle_{OBS}$, i.e. the $sorts(\Sigma)$-sorted congruence $\cong$ such that for any $s \in sorts(\Sigma)$ and any $a, a' \in |\langle A \rangle_{OBS}|_s$, $a \cong_s a'$ if and only if for all contexts $\Gamma \in T_\Sigma(Y \cup \{x{:}s\})$ of an observable sort, where Y is a set of variables of observable sorts, and all valuations $v : Y \to |\langle A \rangle_{OBS}|_{OBS}$,

$$\Gamma(a)_{\langle A \rangle_{OBS}}(v) = \Gamma(a')_{\langle A \rangle_{OBS}}(v).$$

(Here, $\Gamma(a)_{\langle A \rangle_{OBS}}(v)$ stands for $\Gamma_{\langle A \rangle_{OBS}}(\hat{v})$, where $\hat{v}$ is the extension of v to $Y \cup \{x{:}s\}$ given by $\hat{v}(x) = a$, and similarly for $\Gamma(a')_{\langle A \rangle_{OBS}}(v)$.) It is easy to see that the relation $\cong$ so defined is a congruence, and moreover, that it is the identity on observable sorts.

3. Define Z as the quotient of $\langle A\rangle_{OBS}$ by $\cong$.

It easily follows from the above construction that $A \equiv_{OBS} Z$. So, to complete the proof it is enough to show that $Z \models \Phi$. Suppose that this does not hold. That is, for some $\varphi \in \Phi$ of the form

$$\forall X.\ (\bigwedge_{i \in I} t_i = t'_i) \Rightarrow t = t'$$

there is a valuation $v : X \to |Z|$ such that $(t_i)_Z(v) = (t'_i)_Z(v)$ for all $i \in I$, but $t_Z(v) \neq t'_Z(v)$. By the construction of Z, there is a set Y of variables of observable sorts, a valuation $w : Y \to |Z|$ and a substitution $\theta : X \to T_\Sigma(Y)$ such that for $x \in X$, $\theta(x)_Z(w) = v(x)$. Then for all $i \in I$, $(t_i[\theta])_Z(w) = (t'_i[\theta])_Z(w)$, and $(t[\theta])_Z(w) \neq (t'[\theta])_Z(w)$. Hence, there exists a context $\Gamma \in T_\Sigma(Y1 \cup \{x{:}s\})$ and valuation $u : Y1 \to |Z|$, where $Y1$ is a set of variables of observable sorts (we can assume that Y and $Y1$ are disjoint), such that $\Gamma((t[\theta])_Z(w))_A(u) \neq \Gamma((t'[\theta])_Z(w))_A(u)$. Let $w \cup u : Y \cup Y1 \to |Z|$ be the union of the valuations w and u. Then, in the algebra A, for all $i \in I$, $(t_i[\theta])_A(w \cup u) = (t'_i[\theta])_A(w \cup u)$, and $\Gamma(t[\theta])_A(w \cup u) \neq \Gamma(t'[\theta])_A(w \cup u)$. That is, A does not satisfy the following observable consequence of φ:

$$\forall Y \cup Y1.\ (\bigwedge_{i \in I} t_i[\theta] = t'_i[\theta]) \Rightarrow \Gamma(t[\theta]) = \Gamma(t'[\theta])$$

which contradicts the assumption that $A \models ObsCon(\Phi)$. □

The condition that the axioms in $SP2$ have observable premises is essential here; there are well-known examples of conditional equational specifications with non-observable premises which do not have a terminal model as used in the above proof. The same assumption appears in [GM 82] and [BW 82]; we can see no way to avoid this either.

An important special case of the situation when $SP2$ is given as a list of conditional equations with observable premises is when there are no premises at all, i.e. when $SP2$ is given as a list of equations.

Corollary 3.6 *Consider two specifications $SP1$ and $SP2$ over the same algebraic signature Σ and a set $OBS \subseteq sorts(\Sigma)$ of observable sorts. Suppose that $SP2$ is given as a set Φ of Σ-equations. Then $SP1 \models^{OBS} SP2$ iff $SP1 \models ObsCon(\Phi)$.* □

In practice it is often the case that some of the axioms of the specification $SP2$ may be proved directly from the specification $SP1$. Then there is no need to look at their observable consequences:

Corollary 3.7 *Consider two specifications $SP1$ and $SP2$ over the same algebraic signature Σ and a set $OBS \subseteq sorts(\Sigma)$ of observable sorts. Suppose that $SP2$ is given as a set Φ of conditional Σ-equations with observable premises. Let $\Phi = \Phi1 \cup \Phi2$. Then, if $SP1 \models \Phi1$ and $SP1 \models ObsCon(\Phi2)$ then $SP1 \models^{OBS} SP2$.*

Proof Trivially follows from Theorem 3.5, since observable consequences are consequences in the usual sense. □

Counterexample The assumption that all the premises in the conditional axioms in Φ are of observable sorts is essential in Corollary 3.7, i.e. we cannot allow conditional equations with non-observable premises even in $\Phi1$. Consider:

$$
\begin{array}{rcl}
\Sigma & = & \Sigma\mathtt{Bool} \cup \\
& & \mathtt{sorts}\ \ \mathtt{s}, \mathtt{obs} \\
& & \mathtt{opns}\ \ \ \mathtt{a}, \mathtt{b} : \mathtt{s} \\
& & \qquad\quad \mathtt{c}, \mathtt{d} : \mathtt{obs} \\
& & \qquad\quad \mathtt{f} : \mathtt{obs} \to \mathtt{s}
\end{array}
$$

$$OBS = \{\mathtt{obs}, \mathtt{bool}\}$$

$$
\begin{array}{rcl}
\Phi 1 & = & \{\mathtt{a} = \mathtt{b} \Rightarrow \mathtt{true} = \mathtt{false}\} \\
\Phi 2 & = & \{\mathtt{f}(\mathtt{c}) = \mathtt{a}, \mathtt{f}(\mathtt{d}) = \mathtt{b}\}
\end{array}
$$

Suppose now that $SP1$ is a Σ-specification which ensures that all its models interpret the Boolean part in the standard way, and moreover, in all models of $SP1$ the equality $\mathtt{a} = \mathtt{b}$ does not hold. But suppose that there are models of $SP1$ in which $\mathtt{c} = \mathtt{d}$ holds.

Then, $SP1 \models \Phi 1$ and also $SP1 \models ObsCon(\Phi 2)$, since there are no observable consequences of the equations in $\Phi 2$. However, $SP1 \not\models^{OBS} \Phi 1 \cup \Phi 2$: if B is a model of $SP1$ such that $\mathtt{c}_B = \mathtt{d}_B$ then no model of $\Phi 1 \cup \Phi 2$ is behaviourally equivalent to B. $\square$

As we mentioned earlier, checking that an Extended ML structure STR fits an Extended ML signature SIG involves proving behavioural consequence between two specifications over different algebraic signatures. According to the Standard ML matching rules, STR may contain more components then SIG, hence the algebraic signature Σ_{STR} associated with STR may be larger than the algebraic signature Σ_{SIG} associated with SIG. Moreover, because STR is permitted to share more than SIG requires, the real requirement is that a quotient of Σ_{SIG} is a sub-signature of Σ_{STR}. It is important to decide which of the two algebraic signatures will provide the operations we can use to build observable terms. It turns out that the appropriate choice is almost always Σ_{SIG}.

Definition 3.8 Given two specifications $SP1$ and $SP2$, an algebraic signature morphism $\sigma: Sig[SP2] \to Sig[SP1]$, and a set of sorts $OBS \subseteq sorts(Sig[SP2])$, we say that $SP2$ is a *behavioural consequence of SP1 with respect to OBS via* σ, written $SP1 \models_\sigma^{OBS} SP2$, if

$$\textbf{derive from } SP1 \textbf{ by } \sigma \models^{OBS} SP2$$

where, as in [ST 88a], for any specification SP' and $\sigma: \Sigma \to Sig[SP']$, **derive from** SP' **by** σ is a specification with semantics given by:

$$
\begin{array}{rcl}
Sig[\textbf{derive from } SP' \textbf{ by } \sigma] & = & \Sigma \\
Mod[\textbf{derive from } SP' \textbf{ by } \sigma] & = & \{A'|_\sigma \mid A' \in Mod[SP']\}
\end{array}
$$

where $A'|_\sigma$ is the σ-reduct of the algebra A'.

Another possibility would be to consider a set of observable sorts $OBS' \subseteq sorts(Sig[SP1])$, and define

$$SP1 \models_\sigma^{OBS'} SP2 \iff SP1 \models^{OBS'} \textbf{translate } SP2 \textbf{ by } \sigma$$

where, as in [ST 88a], for any specification SP and $\sigma: Sig[SP] \to \Sigma'$, **translate** SP **by** σ is a specification with semantics given by:

$$
\begin{array}{rcl}
Sig[\textbf{translate } SP \textbf{ by } \sigma] & = & \Sigma' \\
Mod[\textbf{translate } SP \textbf{ by } \sigma] & = & \{A \in Alg(\Sigma') \mid A|_\sigma \in Mod[SP]\}.
\end{array}
$$

We have chosen the more permissive of the two possibilities:

Fact 3.9 *Given two specifications SP1 and SP2, an algebraic signature morphism* $\sigma\colon Sig[SP2] \to Sig[SP1]$*, and a set of sorts* $OBS \subseteq sorts(Sig[SP2])$*,*

$$\text{if} \qquad SP1 \models^{\sigma(OBS)} \textbf{translate } SP2 \textbf{ by } \sigma \qquad \text{then} \qquad \textbf{derive from } SP1 \textbf{ by } \sigma \models^{OBS} SP2.$$

Proof Let $A1 \in Mod[SP1]$. Since $SP1 \models^{\sigma(OBS)}$ **translate** $SP2$ **by** σ, there exists $B1 \in Alg(Sig[SP1])$ such that $B1 \equiv_{\sigma(OBS)} A1$ and $B1 \in Mod[$**translate** $SP2$ **by** $\sigma]$, that is, $B1|_\sigma \in Mod[SP2]$. Since reduct functors preserve behavioural equivalence (this follows from Fact 5 of [ST 87]), $A1|_\sigma \equiv_{OBS} B1|_\sigma$, so $A1|_\sigma \models^{OBS} SP2$, which completes the proof. □

Notation In the rest of this paper we write $SP1 \models^{OBS}_{Sig[SP2]} SP2$ or even $SP1 \models^{OBS} SP2$ since σ will be unambiguously determined by the context and the way that names are handled in the semantics of Extended ML. We use a similar convention for individual algebras. If $A \in Alg(Sig[SP1])$, we write $A \models^{OBS} SP2$ to denote $A|_\sigma \models^{OBS} SP2$.

The following well-known fact (proved for equations in [BG 80] and for sentences of first-order equational logic in [GB 84]) allows us to use Proposition 3.4, Theorem 3.5 and Corollaries 3.6 and 3.7 to prove behavioural consequence between specifications over different signatures as well:

Lemma 3.10 (Satisfaction Lemma) *If* $\sigma\colon \Sigma \to \Sigma'$ *is a signature morphism,* φ *is a closed* Σ*-formula and* A' *is a* Σ'*-algebra, then* $A' \models \sigma(\varphi)$ *iff* $A'|_\sigma \models \varphi$*.* □

As a consequence of this, Proposition 3.4 may be used to check that $SP1 \models^{OBS} SP2$ (notation as in the above definition), if *SP*2 is given by a list of axioms over *Sig*[*SP*2]. This is always the case if *SP*2 is an Extended ML signature without hidden types or values. We translate these axioms to *Sig*[*SP*1] and show that they hold in any model of *SP*1. Similarly, Theorem 3.5 and its corollaries may be used by translating the observable consequences of axioms in *SP*2 to *Sig*[*SP*1] and proving that they hold in any model of *SP*1. Our definition allows the set of observable consequences of axioms in *SP*2 to be formed in $Sig[SP2]$ which is more permissive than forcing them to be formed in $Sig[SP1]$ after translating the axioms (this would correspond to the other choice for behavioural consequence mentioned above).

This is just a special case (but an important one) of the general problem of proving behavioural consequence between structured specifications. For example, the above comments do not apply directly to the situation when $SP2$ has a non-trivial structure. More on this topic may be found in [Far 89]. Appendix A contains two technical lemmas which will be used later to prove satisfaction up to behavioural equivalence in some important specific situations.

4 Semantics of functors

4.1 Standard ML functors

Consider a Standard ML functor

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

The Standard ML signatures SIG_{in} and SIG_{out} determine algebraic signatures Σ_{in} and Σ_{out}, respectively. These are not disjoint in general. Their common part $\Sigma_{shr} = \Sigma_{in} \cap \Sigma_{out}$ with signature inclusions

$\iota_{in} : \Sigma_{shr} \hookrightarrow \Sigma_{in}$ and $\iota_{out} : \Sigma_{shr} \hookrightarrow \Sigma_{out}$ expresses the sharing requirements *sharing-decl* in the functor heading. The internal names used in Σ_{out} other than those inherited from Σ_{in} are new.

Since *BODY* is just Standard ML code, it determines the *basic semantics* of the functor F as a function $F_{bsem} : Alg(\Sigma_{in}) \to Alg(\Sigma_{out})$ which for any algebra $A \in Alg(\Sigma_{in})$, builds an algebra $F_{bsem}(A) \in Alg(\Sigma_{out})$ such that $F_{bsem}(A)\big|_{\Sigma_{shr}} = A\big|_{\Sigma_{shr}}$.

The complete picture is a bit more complex. The argument for F may be a much larger structure STR_{arg} with algebraic signature Σ_{arg}, which may in addition contain more sharing than required by the functor input signature. The matching rules of the language (which are the same for Extended ML as for Standard ML) will determine an algebraic signature morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$. Any identification σ makes on Σ_{in} must be preserved when the functor F is applied to STR_{arg}. The following technicalities capture this idea.

For any algebraic signature morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$, the *translation of* Σ_{arg} *by* F *via* σ, written $F(\Sigma_{arg}[\sigma])$, and the *translation of* σ *by* F, written $F[\sigma]$, are such that the following diagram

$$\begin{tikzcd}
\Sigma_{arg} \arrow[r, hook, "\iota'_{out}"] & F(\Sigma_{arg}[\sigma]) \\
\Sigma_{in} \arrow[u, "\sigma"] & \\
\Sigma_{shr} \arrow[u, hook, "\iota_{in}"] \arrow[r, hook, "\iota_{out}"'] & \Sigma_{out} \arrow[uu, "F[\sigma]"']
\end{tikzcd}$$

is a pushout in the category of algebraic signatures (where all the hooked arrows represent algebraic signature inclusions).

For those who dislike the simplicity of the language of category theory, let us point out that the signature $F(\Sigma_{arg}[\sigma])$ may be constructed as the disjoint union of the signature Σ_{arg} and the difference $(\Sigma_{out} \setminus \Sigma_{shr})$ with ranks of operations renamed accordingly. Then $F[\sigma]$ is the union of the inclusion of $(\Sigma_{out} \setminus \Sigma_{in})$ into $F(\Sigma_{arg}[\sigma])$ and the morphism σ restricted to Σ_{shr}.

Any Σ_{arg}-algebra A may be "fitted" as an argument for the functor F using the morphism σ: namely, $A\big|_\sigma$ is a Σ_{in}-algebra to which we can apply F_{bsem}. The requirement on F_{bsem} ensures that $F_{bsem}(A\big|_\sigma)\big|_{\Sigma_{shr}} = (A\big|_\sigma)\big|_{\Sigma_{shr}}$. Thus, there exists a unique $F(\Sigma_{arg}[\sigma])$-algebra $F_{gres}(A[\sigma])$ (the amalgamated union of A and $F_{bsem}(A\big|_\sigma)$ — cf. [EM 85], [ST 88b]) such that

- $F_{gres}(A[\sigma])\big|_{\Sigma_{arg}} = A$, and
- $F_{gres}(A[\sigma])\big|_{F[\sigma]} = F_{bsem}(A\big|_\sigma)$.

We refer to the $F(\Sigma_{arg}[\sigma])$-algebra $F_{gres}(A[\sigma])$ as the *global result* of the application of F to A (along the fitting morphism σ).

Again, $F_{gres}(A[\sigma])$ may be constructed more explicitly by combining its components in A and $F_{bsem}(A\big|_\sigma)$.

The global result of functor application is "larger" than indicated in Section 2. We expect a structure over the output signature as a result. However, as mentioned above, the sharing between those components of the actual parameter that occur in the output must be preserved. Thus, the *result* of applying F to A (along the fitting morphism σ), written $F_{res}(A[\sigma])$, is the reduct $F_{gres}(A[\sigma])\big|_{F[\sigma](\Sigma_{out})}$

of the global result to the signature $F[\sigma](\Sigma_{out})$ which is the image of the output signature Σ_{out} under the signature morphism $F[\sigma]$.

In the above presentation of functor semantics, we have adopted what may be thought of as a "local" view of the algebraic signature Σ_{arg} and algebra A, in which they model the structure to which the functor is actually applied. There is an alternative "global" view, suggested by the fact that we develop a modular Standard ML program by defining a collection of interrelated structures. The resulting structure environment may be viewed as a single structure having all the top-level structures as substructures. Its algebraic signature is the union of the algebraic signatures of the individual structure components, and the algebra it denotes is the amalgamated union of the algebras denoted by the components. With this in mind we may interpret the algebra A and its algebraic signature Σ_{arg} in the above as representing this whole structure. It seems to be necessary to adopt this view since sharing may take place between two separate structures in the environment, and thus some structure which is not included in the actual parameter explicitly passed to a functor may nonetheless provide some additional means of manipulating values of the shared types.

4.2 Extended ML functors

The semantics of Standard ML functors in the previous subsection may be carried over to Extended ML functors as well, but we have to cope with a few additional issues.

Consider an Extended ML functor

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl} = BODY$$

Recall that SIG_{in}, SIG_{out} and $BODY$ may contain axioms, and so are in fact specifications with algebraic signatures $\Sigma_{in} = Sig[SIG_{in}]$, $\Sigma_{out} = Sig[SIG_{out}]$ and $\Sigma_{body} = Sig[BODY]$, respectively, and classes of models $Mod[SIG_{in}] \subseteq Alg(\Sigma_{in})$, $Mod[SIG_{out}] \subseteq Alg(\Sigma_{out})$ and $Mod[BODY] \subseteq Alg(\Sigma_{body})$, respectively. The comments in the previous subsection concerning the relationship between Σ_{in} and Σ_{out} still apply. Moreover, we have an algebraic signature morphism $\tau : \Sigma_{out} \to \Sigma_{body}$ (this allows the body to contain more components than required by the output signature and for extra sharing between the components the output will contain) and an algebraic signature inclusion $\iota_b : \Sigma_{shr} \hookrightarrow \Sigma_{body}$ such that the following diagram commutes:

$$\begin{tikzcd}
\Sigma_{body} & \\
\Sigma_{shr} \arrow[u, hook, "\iota_b"] \arrow[r, hook, "\iota_{out}"'] & \Sigma_{out} \arrow[ul, "\tau"']
\end{tikzcd}$$

(the only way that the output can share with the input is via the body). Note that, as in Standard ML, the input is not automatically included in the body and so $\Sigma_{in} \cap \Sigma_{body}$ may be a proper subsignature of Σ_{in} (but by the above assumption, Σ_{body} has to contain $\Sigma_{shr} = \Sigma_{in} \cap \Sigma_{out}$).

As with Standard ML functors, we require that the shared part of the input is preserved by the body. In particular if Σ_{in} is a subsignature of Σ_{out} then this constraint means that we force the basic semantics of Extended ML functors to be *persistent* [EM 85]. This was a trivial requirement for Standard ML functors since Standard ML code does not allow the programmer to modify the input. In Extended ML, however, this may lead to inconsistency since the body may impose new requirements on the input.

The basic semantics $\mathcal{F}_{bsem} : Alg(\Sigma_{in}) \rightarrow \mathcal{P}ow(Alg(\Sigma_{out}))$ of the above Extended ML functor assigns to any Σ_{in}-algebra a class of Σ_{out}-algebras determined by $BODY$ such that for any $A \in Alg(\Sigma_{in})$:

$$\mathcal{F}_{bsem}(A) = \{B|_\tau \mid B \in Mod[BODY] \text{ and } B|_{\Sigma_{in}\cap\Sigma_{body}} = A|_{\Sigma_{in}\cap\Sigma_{body}}\}.$$

The *domain* of F is defined as follows:

$$Dom(F) = \{A \in Alg(\Sigma_{in}) \mid \mathcal{F}_{bsem}(A) \neq \emptyset\}.$$

In the following, we will identify the function $\mathcal{F}_{bsem}$ with the family of all the partial functions $F_{bsem} : Alg(\Sigma_{in}) \rightsquigarrow Alg(\Sigma_{out})$ such that $F_{bsem}(A)$ is defined exactly when $A \in Dom(F)$ and then $F_{bsem}(A) \in \mathcal{F}_{bsem}(A)$. Hence, $F_{bsem}(A)$ will stand for an arbitrary algebra in $\mathcal{F}_{bsem}(A)$. We will refer to both $\mathcal{F}_{bsem}$ and any F_{bsem} as *basic semantic functions*, where the context and the font will determine which notion is being used.

Note that for any $F_{bsem} \in \mathcal{F}_{bsem}$ and $A \in Dom(F)$, $A|_{\Sigma_{shr}} = F_{bsem}(A)|_{\Sigma_{shr}}$, as with Standard ML functors. In fact, if $BODY$ contains only Standard ML code, then the family $\mathcal{F}_{bsem}$ has exactly one element F_{bsem}, which is the basic semantics of the corresponding Standard ML functor. Thus, the above definition of the basic semantics of Extended ML functors properly generalises the basic semantics of Standard ML functors. The only difference is that in Extended ML the code need not be executable, and it need not define the result unambiguously (it may even be inconsistent, in which case no result exists). Just as before, we can extend each of the basic semantic functions $F_{bsem} \in \mathcal{F}_{bsem}$ to the partial semantic functions F_{gres} and F_{res} operating on any algebra matching the input signature. The result of applying F to an Extended ML structure STR matching Σ_{in} via a fitting morphism $\sigma : \Sigma_{in} \rightarrow Sig[STR]$ determined by the ML matching rules is a specification with semantics defined "pointwise":

$$\begin{aligned} Sig[F(STR)] &= F[\sigma](\Sigma_{out}) \\ Mod[F(STR)] &= \{F_{res}(A[\sigma]) \mid A \in Mod[STR] \text{ and } F_{bsem} \in \mathcal{F}_{bsem}\}. \end{aligned}$$

Notation For any Extended ML functor

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

we use all the notation introduced above without recalling it explicitly. For example, Σ_{in} will always denote $Sig[SIG_{in}]$, Σ_{shr} will denote $\Sigma_{in} \cap \Sigma_{out}$, $\mathcal{F}_{bsem}$ is the family of basic semantic functions defined above, and so on. We will also feel free to modify the above notation by using indices, primes, etc.

The above basic semantics of Extended ML functors completely disregards the fact that signatures may contain axioms and indeed takes account only of the axioms given in the functor body. Axioms in structures and functor bodies in Extended ML play the role of non-executable code. This is in contrast with the axioms in the signatures, which are important only as specifications of the (executable or not) code. Rather than take them into consideration when defining the above "operational" semantics of functors, we introduce a notion of correctness meant to model the intuitive idea that functors should fulfill the requirements stated in their headings.

For a functor to be correct we will require that if the input structure satisfies the requirements imposed by the input signature then the functor produces result structure(s) which satisfy the requirements stated in the output signature. As we have indicated previously, axioms in signatures should be considered only up to behavioural equivalence w.r.t. a pre-specified set of primitive types that the user may directly observe. In Extended ML we take those to be exactly the built-in pervasive types $sorts(\texttt{Perv})$ (with their interpretation inherited from Standard ML). We require that the structure produced by applying a functor to a given input structure satisfies the output requirements not necessarily literally but only up to behavioural equivalence. Consequently, however, we have to accept the possibility that the requirements in the input signature are not satisfied literally, but again only up to behavioural equivalence. The reader should be warned here against interpreting this statement in an oversimplified manner: it is not enough to consider the input and output signature separately from contexts in which the functor may potentially be used[7]. Looking just at the input or output signature as it stands yields very few non-trivial "observations" (terms of primitive types) for most of the types in the signature. However, when the functor is used and the input types are instantiated in a richer context, the user usually has many more ways to observe the types of the resulting structure. Thus, behavioural equivalence must be considered at a global level: at the level of the environment in which the actual input structure resides and to which the result structure is added. The global view of functor parameters (see the discussion following the semantics of Standard ML functors in the previous subsection) provides an appropriate framework to formalise these ideas. The following definition follows almost directly the notion of *universal implementation* of [Sch 86].

Definition 4.1 An Extended ML functor of the form

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl} = BODY$$

is *universally correct* if for any algebraic (argument) signature Σ_{arg} and fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$, any Σ_{arg}-algebra A such that $A \models^{sorts(\texttt{Perv})}$ **translate** SIG_{in} **by** σ and any $F_{bsem} \in \mathcal{F}_{bsem}$:

1. $A|_\sigma \in Dom(F)$;
2. $F_{gres}(A[\sigma]) \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $F[\sigma]$; and
3. for any Σ_{arg}-algebra B such that $B \equiv_{sorts(\texttt{Perv})} A$ and $B|_\sigma \models SIG_{in}$, there exists a $F(\Sigma_{arg}[\sigma])$-algebra $\widehat{B}$ such that $\widehat{B}|_{\iota'_{out}} = B$, $\widehat{B} \equiv_{sorts(\texttt{Perv})} F_{gres}(A[\sigma])$ and $\widehat{B}|_{F[\sigma]} \models SIG_{out}$.

A careful reader may have realized that condition 3 entails condition 2 (and more implicitly, condition 1 as well). We have stated these conditions separately since conditions 1 and 2 are what one intuitively expects while condition 3 turns out to be required for technical reasons in situations in which a programming task is decomposed into separate but interacting subtasks (see for example the proof of Proposition 5.3 in Appendix D).

In our methodology, a programming task is presented as an Extended ML functor heading. The programmer is to produce a functor body consisting of Standard ML code such that the functor is universally correct. In the rest of this paper we present some methods for achieving this goal by modular decomposition (Section 5) and stepwise refinement (Section 6) with explicit conditions which ensure the correctness of the result.

[7] In fact, in [ST 85b] we have proposed a semantics for Extended ML based on such a view of functors and signatures as "closed" entities. We now consider this to be a methodological mistake and propose a different view, better suited as a basis for the methodology we develop.

4.3 Multi-argument functors

In the previous subsection we have only considered functors having a single argument. The case of multiple arguments is a little more complicated, but as in Standard ML any functor with multiple arguments may be reduced to a one-argument functor by combining the input signatures into a single signature. Below are the technicalities for the case of a two-argument functor of the following form:

$$\texttt{functor}\ G(Y1 : SIG1_{in}, Y2 : SIG2_{in}\ \texttt{sharing}\ \textit{sharing-constr}) : SIG_{out}\\ \texttt{sharing}\ \textit{sharing-decl},$$

Arbitrary multi-argument functors may be handled in a similar way.

Let $\Sigma1_{in}$, $\Sigma2_{in}$ and Σ_{out} be algebraic signatures corresponding to $SIG1_{in}$, $SIG2_{in}$ and SIG_{out}, respectively. As before, the sharing declarations in the functor heading force them to overlap. Define $\Sigma_{com} = \Sigma1_{in} \cap \Sigma2_{in}$, $\Sigma1_{shr} = \Sigma1_{in} \cap \Sigma_{out}$ and $\Sigma2_{shr} = \Sigma2_{in} \cap \Sigma_{out}$ with algebraic signature inclusions as indicated in the following diagram:

$$\begin{tikzcd}
\Sigma_{com} \arrow[rr, "\iota 2_{com}"] \arrow[dd, "\iota 1_{com}"'] & & \Sigma2_{in} & \\
 & & & \Sigma2_{shr} \arrow[ul, "\iota 2_{in}"'] \arrow[dd, "\iota 2_{out}"] \\
\Sigma1_{in} & & & \\
 & \Sigma1_{shr} \arrow[ul, "\iota 1_{in}"] \arrow[rr, "\iota 1_{out}"'] & & \Sigma_{out}
\end{tikzcd}$$

The basic semantics of G maps any two algebras $A1$ and $A2$, over algebraic signatures $\Sigma1_{in}$ and $\Sigma2_{in}$ respectively, to a Σ_{out}-algebra. But since the heading of G requires its arguments to share on Σ_{com}, this basic semantics is defined only for algebras $A1$ and $A2$ that coincide on Σ_{com}. Moreover, the components shared between input and output must be preserved, as discussed in the previous subsection for one-argument functors. Thus, the basic semantics of G is a function $\mathcal{G}^2_{bsem}$ defined on

$$\{\langle A1, A2\rangle \mid A1 \in Alg(\Sigma1_{in}), A2 \in Alg(\Sigma2_{in}), A1|_{\Sigma_{com}} = A2|_{\Sigma_{com}}\}$$

such that for any $A1 \in Alg(\Sigma1_{in})$ and $A2 \in Alg(\Sigma2_{in})$ satisfying $A1|_{\Sigma_{com}} = A2|_{\Sigma_{com}}$, $\mathcal{G}^2_{bsem}(A1, A2)$ is a class of Σ_{out}-algebras such that for any $G^2_{bsem}(A1, A2) \in \mathcal{G}^2_{bsem}(A1, A2)$ (recall the convention of identifying $\mathcal{F}_{bsem}$ with a family of partial functions F_{bsem}),

$$G^2_{bsem}(A1, A2)|_{\Sigma1_{shr}} = A1|_{\Sigma1_{shr}} \qquad \text{and} \qquad G^2_{bsem}(A1, A2)|_{\Sigma2_{shr}} = A2|_{\Sigma2_{shr}}.$$

Alternatively, G may be viewed as a one-argument functor by combining $SIG1_{in}$ and $SIG2_{in}$ into a single signature. This combined signature must incorporate the sharing required by *sharing-constr* so that algebras over the corresponding algebraic signature correspond exactly to pairs of algebras which coincide on Σ_{com}. Consider the following diagram:

$$
\begin{tikzcd}
\Sigma_{com} \arrow[rr, "\iota 2_{com}"] \arrow[dd, "\iota 1_{com}"'] & & \Sigma 2_{in} \arrow[dl, "\iota 1'_{com}"] & & \\
& \Sigma_{in} & & & \\
\Sigma 1_{in} \arrow[ur, "\iota 2'_{com}"'] & & & & \Sigma 2_{shr} \arrow[uull, "\iota 2_{in}"'] \arrow[dl, "\iota 1_{shr}"'] \arrow[dd, "\iota 2_{out}"] \\
& & & \Sigma_{shr} \arrow[uull, "\iota_{in}"] \arrow[dr, "\iota_{out}"] & \\
& & \Sigma 1_{shr} \arrow[uull, "\iota 1_{in}"] \arrow[ur, "\iota 2_{shr}"] \arrow[rr, "\iota 1_{out}"'] & & \Sigma_{out}
\end{tikzcd}
$$

where we require that the two sub-diagrams

$$
\begin{tikzcd}
\Sigma_{com} \arrow[rr, "\iota 2_{com}"] \arrow[dd, "\iota 1_{com}"'] & & \Sigma 2_{in} \arrow[dl, "\iota 1'_{com}"] \\
& \Sigma_{in} & \\
\Sigma 1_{in} \arrow[ur, "\iota 2'_{com}"'] & &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& & \Sigma 2_{shr} \arrow[dl, "\iota 1_{shr}"'] \\
& \Sigma_{shr} & \\
\Sigma 1_{shr} \arrow[ur, "\iota 2_{shr}"] & &
\end{tikzcd}
$$

are (respectively) a pushout and a coproduct in the category of algebraic signatures[8], and where the morphisms $\iota_{in} : \Sigma_{shr} \to \Sigma_{in}$ and $\iota_{out} : \Sigma_{shr} \to \Sigma_{out}$ are defined using the coproduct property of Σ_{shr}.

There is a natural 1–1 correspondence between the partial functions G^2_{bsem} as above and basic semantic functions

$$G_{bsem} : Alg(\Sigma_{in}) \rightsquigarrow Alg(\Sigma_{out})$$

such that for any $A \in Alg(\Sigma_{in})$, if $G_{bsem}(A)$ is defined then $A\big|_{\Sigma_{shr}} = G_{bsem}(A)\big|_{\Sigma_{shr}}$.

In fact, the heading of the functor G may be equivalently rewritten as:

$$\texttt{functor}\ G(Y : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl}^{9}$$

where

$$SIG_{in} =_{def} \textbf{translate}\ SIG1_{in}\ \textbf{by}\ \iota 2'_{com} \cup \textbf{translate}\ SIG2_{in}\ \textbf{by}\ \iota 1'_{com}.$$

We use here the operation of union of specifications over the same algebraic signature formally defined as follows (cf. [ST 88a]): for any specifications SP and SP' such that $Sig[SP] = Sig[SP']$, $SP \cup SP'$ is a specification with semantics given by:

$$
\begin{aligned}
Sig[SP \cup SP'] &= Sig[SP] \qquad (= Sig[SP']) \\
Mod[SP \cup SP'] &= Mod[SP] \cap Mod[SP'].
\end{aligned}
$$

[8] Since we assume that all signatures contain the pervasives of Standard ML which are preserved by all signature morphisms, the coproduct here corresponds to a pushout in the category of algebraic signatures, where the signature of `Perv` is shared.

[9] The sharing declaration here should actually be the one obtained from *sharing-decl* by converting references to $Y1$ and $Y2$ into Y references as appropriate.

Of course, neither union nor **translate** is available in Extended ML, but if $SIG1_{in}$ and $SIG2_{in}$ are Extended ML signatures then it is clear that we can write an Extended ML signature which is equivalent to SIG_{in} defined as above.

Although the two versions of G are equivalent at the level of their basic semantics, and have identical "computational" properties, their correctness properties are not necessarily the same when interfaces are considered up to behavioural equivalence. Given the above situation, for any algebraic signature Σ_{arg}, fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$ and Σ_{arg}-algebra A:

$$
\begin{array}{ll}
\text{if} & A \models_{\Sigma_{arg}}^{sorts(\mathtt{Perv})} \textbf{translate } SIG_{in} \textbf{ by } \sigma \\
\text{then} & A \models^{sorts(\mathtt{Perv})} \textbf{translate } SIG1_{in} \textbf{ by } \iota 2'_{com};\sigma \\
& \text{and} \\
& A \models^{sorts(\mathtt{Perv})} \textbf{translate } SIG2_{in} \textbf{ by } \iota 1'_{com};\sigma.
\end{array}
$$

The opposite implication does not hold in general. If there are non-observable sorts shared by $\Sigma1_{in}$ and $\Sigma2_{in}$, then A may satisfy both $SIG1_{in}$ and $SIG2_{in}$ separately up to behavioural equivalence without satisfying them "jointly" up to behavioural equivalence. In fact, A may satisfy $SIG1_{in}$ and $SIG2_{in}$ separately up to behavioural equivalence when SIG_{in} is inconsistent!

This gives two possible notions of universal correctness of multi-argument functors. It seems appropriate to choose the weaker of the two, which puts more restrictions on the admissible input by requiring that it satisfies the two components of the input signature jointly. Thus we define a multi-argument functor to be universally correct if its one-argument version constructed as above is universally correct.

5 System design: functor decomposition

In the next two sections we discuss how to develop functors which are universally correct with respect to a given functor heading. In this section we concentrate on defining functors as a composition of simpler functors, i.e. by modular decomposition. The idea is very simple: just come up with a bunch of other functors, and define the functor being implemented as an expression over these functors. Of course, we need to impose appropriate verification conditions to ensure that:

- The expression is well-formed: functors in the expression are always applied to structures whose signatures match their input signatures, and the result signature matches the output signature.
- The functor definition is correct: roughly, for any argument satisfying the input signature, the result produced satisfies the output signature (modulo the discussion concerning behavioural equivalence in Section 4.2).

We will analyze three simple but increasingly complex cases of functor decomposition. For each of these cases we give formal statements of the above informal conditions and prove that they ensure correctness of the functor. We then discuss the general situation in a more sketchy way.

5.1 Unitary decomposition

We begin with the simplest case, when a functor is implemented by directly calling another functor.

Consider an Extended ML functor

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = F1(X)$$

where $F1$ is a functor with heading

$$\texttt{functor}\ F1(X : SIG1_{in}) : SIG1_{out}\ \texttt{sharing}\ \mathit{sharing\text{-}decl}1$$

This defines the basic semantics of F as roughly the same as that of $F1$ (see the proof of the theorem in Appendix B for details).

To ensure that the definition of F is correct, we have to verify that two conditions are satisfied:

- The definition of F is well-formed according to the Standard ML typechecking rules:
  - A quotient of $\Sigma 1_{in}$ is a subsignature of Σ_{in} (given by a morphism $\tau_{in} : \Sigma 1_{in} \to \Sigma_{in}$).
  - A quotient of Σ_{out} is a subsignature of $\Sigma 1_{out}$ (given by a morphism $\tau_{out} : \Sigma_{out} \to \Sigma 1_{out}$).
  - The sharing between Σ_{in} and Σ_{out} follows from the sharing between $\Sigma 1_{in}$ and $\Sigma 1_{out}$ (as indicated by a morphism $\rho : \Sigma_{shr} \to \Sigma 1_{shr}$).

  This gives rise to the following commutative diagram:

$$\begin{tikzcd}
\Sigma_{in} & & & \\
& \Sigma 1_{in} \arrow[ul, "\tau_{in}"'] & & \\
& \Sigma 1_{shr} \arrow[u, "\iota 1_{in}"] \arrow[r, "\iota 1_{out}"'] & \Sigma 1_{out} & \\
\Sigma_{shr} \arrow[uuu, "\iota_{in}"] \arrow[ur, "\rho"'] \arrow[rrr, "\iota_{out}"'] & & & \Sigma_{out} \arrow[ul, "\tau_{out}"']
\end{tikzcd}$$

- The requirements stated in the functor interfaces match one another:
  - SIG_{in} entails $SIG1_{in}$ up to behavioural equivalence.
  - $SIG1_{out}$ entails SIG_{out} up to behavioural equivalence.

Here is the formal statement of the correctness result:

Theorem 5.1 *Consider Extended ML functors F and $F1$ as above. Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, determining a commutative diagram as above. Suppose the following conditions are satisfied (we use here the notation introduced in Section 3):*

1. $SIG_{in} \models^{sorts(\iota 1_{in}(\rho(\Sigma_{shr})))}_{\Sigma 1_{in}} SIG1_{in}$
2. $SIG1_{out} \models^{sorts(\iota_{out}(\Sigma_{shr}))}_{\Sigma_{out}} SIG_{out}$

Then, if $F1$ is universally correct then F is universally correct.

Proof See Appendix B. $\square$

5.2 Sequential decomposition

Another simple case is when the functor is defined by composing two other functors.

Consider an Extended ML functor

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl} = G2(G1(X))$$

where $G1$ and $G2$ are functors with headings

$$\texttt{functor}\ G1(Y1 : SIG1_{in}) : SIG1_{out}\ \texttt{sharing}\ \textit{sharing-decl}1$$

and

$$\texttt{functor}\ G2(Y2 : SIG2_{in}) : SIG2_{out}\ \texttt{sharing}\ \textit{sharing-decl}2$$

This defines the basic semantics of F as (roughly — see the proof of the theorem in Appendix C for details) the composition of the basic semantics of $G1$ and $G2$.

To ensure that the definition of F is correct, we have to verify that two conditions are satisfied:

- The definition of F is well-formed according to the Standard ML typechecking rules:
  - A quotient of $\Sigma1_{in}$ is a subsignature of Σ_{in} (given by a morphism $\tau_{in} : \Sigma1_{in} \to \Sigma_{in}$).
  - A quotient of $\Sigma2_{in}$ is a subsignature of $\Sigma1_{out}$ (given by a morphism $\tau_{inter} : \Sigma2_{in} \to \Sigma1_{out}$).
  - A quotient of Σ_{out} is a subsignature of $\Sigma2_{out}$ (given by a morphism $\tau_{out} : \Sigma_{out} \to \Sigma2_{out}$).
  - The sharing between Σ_{in} and Σ_{out} follows (by composition) from the sharing between $\Sigma1_{in}$ and $\Sigma1_{out}$ and between $\Sigma2_{in}$ and $\Sigma2_{out}$ (as indicated by morphisms $\rho1 : \Sigma_{shr} \to \Sigma1_{shr}$ and $\rho1 : \Sigma_{shr} \to \Sigma2_{shr}$).

This gives rise to the following commutative diagram:

$$\begin{tikzcd}
\Sigma_{in} & & & & & \\
 & \Sigma1_{in} \arrow[ul, "\tau_{in}"'] & & & & \\
 & \Sigma1_{shr} \arrow[u, "\iota1_{in}"'] \arrow[r, "\iota1_{out}"] & \Sigma1_{out} & & & \\
 & & & \Sigma2_{in} \arrow[ul, "\tau_{inter}"'] & & \\
 & & & \Sigma2_{shr} \arrow[u, "\iota2_{in}"'] \arrow[r, "\iota2_{out}"] & \Sigma2_{out} & \\
\Sigma_{shr} \arrow[uuuuu, "\iota_{in}"] \arrow[uuur, "\rho1"'] \arrow[urrr, "\rho2"'] \arrow[rrrrr, "\iota_{out}"'] & & & & & \Sigma_{out} \arrow[ul, "\tau_{out}"']
\end{tikzcd}$$

- The requirements stated in the functor interfaces match one another:
  - SIG_{in} entails $SIG1_{in}$ up to behavioural equivalence.
  - $SIG1_{out}$ entails $SIG2_{in}$ up to behavioural equivalence.
  - $SIG2_{out}$ entails SIG_{out} up to behavioural equivalence.

Here is the formal statement of the correctness result:

Theorem 5.2 *Consider Extended ML functors F, $G1$ and $G2$ as above. Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, determining a commutative diagram as above. Suppose that the following conditions are satisfied:*

1. $SIG_{in} \models_{\Sigma1_{in}}^{sorts(\iota1_{in}(\rho1(\Sigma_{shr})))} SIG1_{in}$
2. $SIG1_{out} \models_{\Sigma2_{in}}^{sorts(\iota2_{in}(\rho2(\Sigma_{shr})))} SIG2_{in}$
3. $SIG2_{out} \models_{\Sigma_{out}}^{sorts(\iota_{out}(\Sigma_{shr}))} SIG_{out}$

Then, if $G1$ and $G2$ are universally correct then so is F.

Proof See Appendix C. □

5.3 Parallel decomposition

Another simple case of modular decomposition is when part of the task is split into two more or less independent subtasks which are performed by two functors in parallel.

Consider an Extended ML functor

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \mathit{sharing\text{-}decl} = G0(G1(X), G2(X))$$

where $G0$ is a two-argument functor with a heading of the form

$$\texttt{functor}\ G0(Y01 : SIG01_{in}, Y02 : SIG02_{in}\ \texttt{sharing}\ \mathit{sharing\text{-}constr}0) : SIG0_{out}\ \texttt{sharing}\ \mathit{sharing\text{-}decl}0,$$

and $G1$ and $G2$ are functors with headings

$$\texttt{functor}\ G1(Y1 : SIG1_{in}) : SIG1_{out}\ \texttt{sharing}\ \mathit{sharing\text{-}decl}1$$

and

$$\texttt{functor}\ G2(Y2 : SIG2_{in}) : SIG2_{out}\ \texttt{sharing}\ \mathit{sharing\text{-}decl}2$$

For the definition of F to be well-formed, we have to ensure that the appropriate signatures match, that is (recall the notation concerning two-argument functors introduced in Section 4.3):

- A quotient of $\Sigma1_{in}$ is a subsignature of Σ_{in} (given by a morphism $\tau1_{in} : \Sigma1_{in} \to \Sigma_{in}$).
- A quotient of $\Sigma2_{in}$ is a subsignature of Σ_{in} (given by a morphism $\tau2_{in} : \Sigma2_{in} \to \Sigma_{in}$).
- A quotient of $\Sigma01_{in}$ is a subsignature of $\Sigma1_{out}$ (given by a morphism $\tau1_{inter} : \Sigma01_{in} \to \Sigma1_{out}$).

- A quotient of $\Sigma 02_{in}$ is a subsignature of $\Sigma 2_{out}$ (given by a morphism $\tau 2_{inter} : \Sigma 02_{in} \to \Sigma 2_{out}$).
- A quotient of Σ_{out} is a subsignature of $\Sigma 0_{out}$ (given by a morphism $\tau_{out} : \Sigma_{out} \to \Sigma 0_{out}$).

Moreover, we have to make sure that the required sharing between the arguments of $G0$ follows from the sharing information passed by $G1$ and $G2$ (using morphisms $\rho 1_{inter} : \Sigma 0_{com} \to \Sigma 1_{shr}$ and $\rho 2_{inter} : \Sigma 0_{com} \to \Sigma 2_{shr}$). Thus, the matching rules of ML must determine the following commutative diagram in the category of algebraic signatures:

$$
\begin{tikzcd}
\Sigma_{in} & & & & & & & & \\
& & \Sigma 2_{in} \arrow[ull, "\tau 2_{in}"'] & & & & & & \\
& \Sigma 1_{in} \arrow[uul, "\tau 1_{in}"] & & \Sigma 2_{shr} \arrow[ul, "\iota 2_{in}"'] \arrow[rr, "\iota 2_{out}"] & & \Sigma 2_{out} & & & \\
& & \Sigma 1_{shr} \arrow[ul, "\iota 1_{in}"] \arrow[dd, "\iota 1_{out}"'] & & & & \Sigma 02_{in} \arrow[ul, "\tau 2_{inter}"'] & & \\
& & & & \Sigma 0_{com} \arrow[ull, "\rho 1_{inter}"] \arrow[uul, "\rho 2_{inter}"'] \arrow[urr, "\iota 02_{com}"'] \arrow[ddl, "\iota 01_{com}"] & & & & \\
& & \Sigma 1_{out} & & & & & \Sigma 02_{shr} \arrow[uul, "\iota 02_{in}"] \arrow[dd, "\iota 02_{out}"] & \\
& & & \Sigma 01_{in} \arrow[ul, "\tau 1_{inter}"] & & & & & \\
& & & & & \Sigma 01_{shr} \arrow[ull, "\iota 01_{in}"'] \arrow[rr, "\iota 01_{out}"] & & \Sigma 0_{out} & \\
\Sigma_{shr} \arrow[uuuuuuuu, "\iota_{in}"] \arrow[rrrrrrrr, "\iota_{out}"'] & & & & & & & & \Sigma_{out} \arrow[ul, "\tau_{out}"']
\end{tikzcd}
$$

The basic semantics of F, $\mathcal{F}_{bsem} : Alg(\Sigma_{in}) \to \mathcal{P}ow(Alg(\Sigma_{out}))$, is defined as follows: for any Σ_{in}-algebra A,

$$
\begin{aligned}
\mathcal{F}_{bsem}(A) = \{ A0\big|_{\tau_{out}} \mid \quad & A0 \in \mathcal{G}0^2_{bsem}(A1\big|_{\tau 1_{inter}}, A2\big|_{\tau 2_{inter}}) \\
& \text{for some } A1 \in \mathcal{G}1_{bsem}(A\big|_{\tau 1_{in}}) \text{ and } A2 \in \mathcal{G}2_{bsem}(A\big|_{\tau 2_{in}}) \}.
\end{aligned}
$$

Omitting all the problems of definedness of the partial functions involved,[10] any choice of basic semantic functions $G1_{bsem}$, $G2_{bsem}$ and $G0^2_{bsem}$ for $G1$, $G2$ and $G0$ respectively, determines a basic semantic function $F_{bsem} : Alg(\Sigma_{in}) \rightsquigarrow Alg(\Sigma_{out})$ as follows: for any Σ_{in}-algebra A,

$$
F_{bsem}(A) = G0^2_{bsem}(\, G1_{bsem}(A\big|_{\tau 1_{in}})\big|_{\tau 1_{inter}} ,\, G2_{bsem}(A\big|_{\tau 2_{in}})\big|_{\tau 2_{inter}} \,)\big|_{\tau_{out}}.
$$

[10] The verification conditions for this case of decomposition will ensure that this is not a problem.

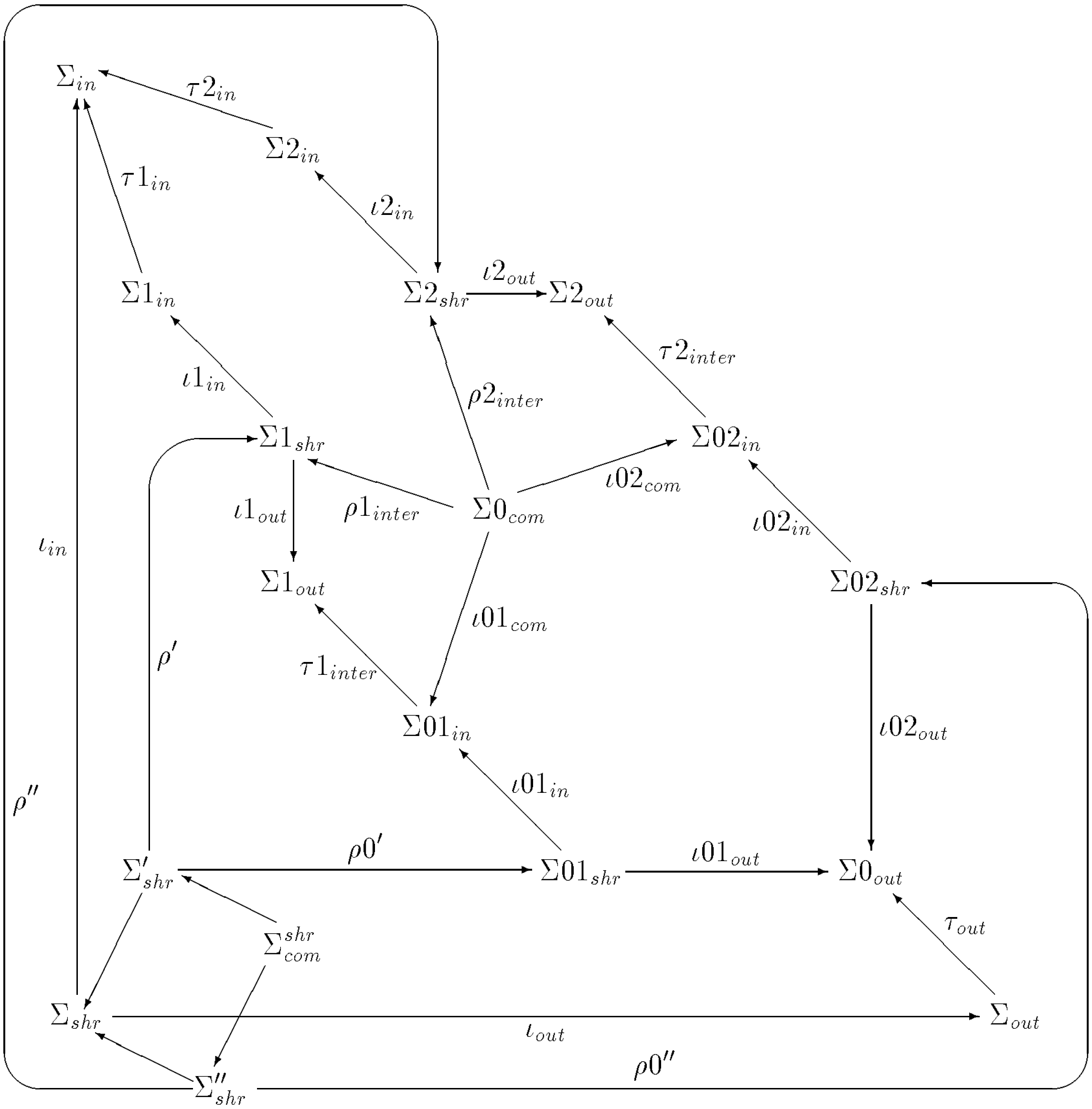
To see that this is well-defined we have to verify that the arguments of $G0^2_{bsem}$ coincide on $\Sigma 0_{com}$:

$$\begin{aligned}
(G1_{bsem}(A\big|_{\tau 1_{in}})\big|_{\tau 1_{inter}})\big|_{\iota 01_{com}} &= ((G1_{bsem}(A\big|_{\tau 1_{in}}))\big|_{\iota 1_{out}})\big|_{\rho 1_{inter}} \\
&= ((A\big|_{\tau 1_{in}})\big|_{\iota 1_{in}})\big|_{\rho 1_{inter}} \\
&= ((A\big|_{\tau 2_{in}})\big|_{\iota 2_{in}})\big|_{\rho 2_{inter}} \\
&= ((G2_{bsem}(A\big|_{\tau 2_{in}}))\big|_{\iota 2_{out}})\big|_{\rho 2_{inter}} \\
&= (G2_{bsem}(A\big|_{\tau 2_{in}})\big|_{\tau 2_{inter}})\big|_{\iota 02_{com}}.
\end{aligned}$$

We still have to ensure the required sharing between Σ_{in} and Σ_{out} (described by the signature Σ_{shr} with algebraic signature inclusions ι_{in} and ι_{out}). Since there are two possible paths by which the output may inherit a part of the input, one via $G1$ and the other via $G2$, the shared subsignature may be split into two (possibly non-disjoint) subsignatures. More formally, there must be a pushout

$$\begin{tikzcd}
\Sigma'_{shr} \arrow[r] & \Sigma_{shr} \\
\Sigma^{shr}_{com} \arrow[u] \arrow[r] & \Sigma''_{shr} \arrow[u]
\end{tikzcd}$$

and two pairs of morphisms, $\rho' : \Sigma'_{shr} \rightarrow \Sigma 1_{shr}$ and $\rho'' : \Sigma''_{shr} \rightarrow \Sigma 2_{shr}$, and $\rho 0' : \Sigma'_{shr} \rightarrow \Sigma 01_{shr}$ and $\rho 0'' : \Sigma''_{shr} \rightarrow \Sigma 02_{shr}$ (determined by the ML matching rules) such that the above diagram augmented by these morphisms commutes. That is:



The commutativity of the above diagram ensures that the semantics of F as defined above indeed satisfies the sharing property: for any $A \in Dom(F)$, $A\big|_{\Sigma_{shr}} = F_{bsem}(A\big|_{\Sigma_{shr}})$.

Recall that ML views the two-argument functor $G0$ as a one-argument functor as defined in Section 4.3. Similarly, we can collapse the two "parallel" functors $G1$ and $G2$.

Let $\Sigma 12_{in}$, $\Sigma 12_{shr}$ and $\Sigma 12_{out}$ be algebraic signatures defined by the following pushouts:

$$\begin{tikzcd}
\Sigma 1_{in} \arrow[r, "c1_{in}"] & \Sigma 12_{in} \\
\Sigma 0_{com} \arrow[u, "\rho 1_{inter};\iota 1_{in}"] \arrow[r, "\rho 2_{inter};\iota 2_{in}"'] & \Sigma 2_{in} \arrow[u, "c2_{in}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\Sigma 1_{shr} \arrow[r, "c1_{shr}"] & \Sigma 12_{shr} \\
\Sigma 0_{com} \arrow[u, "\rho 1_{inter}"] \arrow[r, "\rho 2_{inter}"'] & \Sigma 2_{shr} \arrow[u, "c2_{shr}"']
\end{tikzcd}$$

$$
\begin{tikzcd}
\Sigma 1_{out} \arrow[r, "c1_{out}"] & \Sigma 12_{out} \\
\Sigma 0_{com} \arrow[u, "\rho 1_{inter};\iota 1_{out}"] \arrow[r, "\rho 2_{inter};\iota 2_{out}"'] & \Sigma 2_{out} \arrow[u, "c2_{out}"']
\end{tikzcd}
$$

Define algebraic signature morphisms $\iota 12_{in} : \Sigma 12_{shr} \to \Sigma 12_{in}$ and $\iota 12_{out} : \Sigma 12_{shr} \to \Sigma 12_{out}$ by the pushout property of $\Sigma 12_{shr}$ as the unique morphisms such that

- $c1_{shr};\iota 12_{in} = \iota 1_{in};c1_{in}$ and $c2_{shr};\iota 12_{in} = \iota 2_{in};c2_{in}$, and
- $c1_{shr};\iota 12_{out} = \iota 1_{out};c1_{out}$ and $c2_{shr};\iota 12_{out} = \iota 2_{out};c2_{out}$.

The following diagram may be helpful in visualising the construction so far:

$$
\begin{tikzcd}
\Sigma 12_{in} & \Sigma 2_{in} \arrow[l, "c2_{in}"'] & \Sigma 2_{shr} \arrow[l, "\iota 2_{in}"'] \arrow[dr, "\iota 2_{out}"] \arrow[ddl, "c2_{shr}"] & \\
 & & & \Sigma 2_{out} \arrow[ddl, "c2_{out}"] \\
\Sigma 1_{in} \arrow[uu, "c1_{in}"] & \Sigma 12_{shr} \arrow[uul, "\iota 12_{in}"] \arrow[dr, "\iota 12_{out}"] & & \\
 & & \Sigma 12_{out} & \\
\Sigma 1_{shr} \arrow[uu, "\iota 1_{in}"] \arrow[uur, "c1_{shr}"'] \arrow[dr, "\iota 1_{out}"'] & & & \\
 & \Sigma 1_{out} \arrow[uur, "c1_{out}"'] & & \\
 & & & \Sigma 0_{com} \arrow[uulll, bend left, "\rho 1_{inter}"] \arrow[uuuuuul, bend right, "\rho 2_{inter}"']
\end{tikzcd}
$$

Further, define

$SIG12_{in} =$ **translate** $SIG1_{in}$ **by** $c1_{in} \cup$ **translate** $SIG2_{in}$ **by** $c2_{in}$, and

$SIG12_{out} =$ **translate** $SIG1_{out}$ **by** $c1_{out} \cup$ **translate** $SIG2_{out}$ **by** $c2_{out}$.

(As in Section 4.3, the fact that Extended ML does not include union or **translate** is unimportant since it is clear that we can construct Extended ML signatures which are equivalent to $SIG12_{in}$ and $SIG12_{out}$ as defined above.)

The result of collapsing $G1$ and $G2$ is a functor $G12$ with heading

`functor` $G12(Y12 : SIG12_{in}) : SIG12_{out}$ `sharing` *sharing-decl*$1 \cup$ *sharing-decl*2[11]

and with a body such that the basic semantics $\mathcal{G}12_{bsem}$ is defined by combining the basic semantics of $G1$ and $G2$: for any $\Sigma 12$-algebra A,

$$\mathcal{G}12_{bsem}(A) = \{A12 \in Alg(\Sigma 12_{out}) \mid A12|_{c1_{out}} \in \mathcal{G}1_{bsem}(A|_{c1_{in}}) \text{ and } A12|_{c2_{out}} \in \mathcal{G}2_{bsem}(A|_{c2_{in}})\}.$$

[11]The sharing declaration here should actually be *sharing-decl*$1'$ **and** *sharing-decl*$2'$, where *sharing-decl*$1'$ is obtained from *sharing-decl*1 by converting references to $Y1$ into $Y12$ references as appropriate, and likewise for *sharing-decl*$2'$.

Proposition 5.3 *$G12$ is universally correct provided that $G1$ and $G2$ are.*

Proof See Appendix D. □

It is easy to see that the functor F may be equivalently rewritten as

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = G0(G12(X))$$

with the corresponding algebraic signature diagram

$$\begin{tikzcd}
\Sigma_{in} & & & & & \\
& \Sigma12_{in} \arrow[ul, "\tau_{in}"'] & & & & \\
& \Sigma12_{shr} \arrow[u, "\iota12_{in}"] \arrow[r, "\iota12_{out}"] & \Sigma12_{out} & & & \\
& & & \Sigma0_{in} \arrow[ul, "\tau_{inter}"'] & & \\
& & & \Sigma0_{shr} \arrow[u, "\iota0_{in}"] \arrow[r, "\iota0_{out}"] & \Sigma0_{out} & \\
\Sigma_{shr} \arrow[uuuuu, "\iota_{in}"] \arrow[uuur, "\rho"] \arrow[urrr, "\rho0"'] \arrow[rrrrr, "\iota_{out}"'] & & & & & \Sigma_{out} \arrow[ul, "\tau_{out}"']
\end{tikzcd}$$

where the morphisms $\tau_{in} : \Sigma12_{in} \to \Sigma_{in}$, $\rho : \Sigma_{shr} \to \Sigma12_{shr}$, $\tau_{inter} : \Sigma0_{in} \to \Sigma12_{out}$ and $\rho0 : \Sigma_{shr} \to \Sigma0_{shr}$ are constructed using the pushout properties of their source signatures in the obvious way as the "unions" of, respectively, $\tau1_{in}$ and $\tau2_{in}$, $\rho';c1_{shr}$ and $\rho'';c2_{shr}$, $\tau1_{inter};c1_{out}$ and $\tau2_{inter};c2_{out}$, and $\rho0';\iota01_{shr}$ and $\rho0'';\iota02_{shr}$.

Theorem 5.2 may now be used to prove the universal correctness of F:

Corollary 5.4 *Consider Extended ML functors F, $G0$, $G1$ and $G2$ as above. Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, which determine algebraic signature morphisms as above. Suppose that the following conditions are satisfied:*

1. $SIG_{in} \models_{\Sigma12_{in}}^{sorts(\iota12_{in}(\rho(\Sigma_{shr})))}$ **translate** $SIG1_{in}$ **by** $c1_{in} \cup$ **translate** $SIG2_{in}$ **by** $c2_{in}$

2. **translate** $SIG1_{out}$ **by** $c1_{out} \cup$ **translate** $SIG2_{out}$ **by** $c2_{out} \models_{\Sigma0_{in}}^{sorts(\iota0_{in}(\rho0(\Sigma_{shr})))}$
 translate $SIG01_{in}$ **by** $\iota02'_{com} \cup$ **translate** $SIG02_{in}$ **by** $\iota01'_{com}$

3. $SIG0_{out} \models_{\Sigma_{out}}^{sorts(\Sigma_{shr})} SIG_{out}$

Then, if $G0$, $G1$ and $G2$ are universally correct then so is F.

Proof Follows directly from Theorem 5.2 and Proposition 5.3 (and the definitions of $SIG12_{in}$, $SIG12_{out}$ and $SIG0_{in}$). □

This is not quite satisfactory: the verification conditions of Corollary 5.4 force us to consider some interfaces jointly, even though they are presented separately. The following theorem removes this deficiency. The separate verification conditions must additionally take into account the required sharing between the signatures to which they directly apply and the environment in which the signatures are used. More specifically, since the functors $G1$ and $G2$ are expected to produce overlapping results, the verification conditions must not allow the overlapping parts to be treated (or modified) in different ways.

Theorem 5.5 *Consider Extended ML functors F, $G0$, $G1$ and $G2$ as above. Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, which determine algebraic signature morphisms as above. Suppose that the following conditions are satisfied:*

1. (a) $SIG_{in} \models_{\Sigma 1_{in}}^{sorts(\iota 1_{in}(\rho'(\Sigma'_{shr}))) \cup sorts(\iota 1_{in}(\rho 1_{inter}(\Sigma 0_{com})))} SIG1_{in}$

   (b) $SIG_{in} \models_{\Sigma 2_{in}}^{sorts(\iota 2_{in}(\rho''(\Sigma''_{shr}))) \cup sorts(\iota 2_{in}(\rho 2_{inter}(\Sigma 0_{com})))} SIG2_{in}$

2. (a) $SIG1_{out} \models_{\Sigma 01_{in}}^{sorts(\iota 01_{in}(\rho 0'(\Sigma'_{shr}))) \cup sorts(\iota 01_{com}(\Sigma 0_{com}))} SIG01_{in}$

   (b) $SIG2_{out} \models_{\Sigma 02_{in}}^{sorts(\iota 02_{in}(\rho 0''(\Sigma''_{shr}))) \cup sorts(\iota 02_{com}(\Sigma 0_{com}))} SIG02_{in}$

3. $SIG0_{out} \models_{\Sigma_{out}}^{sorts(\Sigma_{shr})} SIG_{out}$

Then, if $G0$, $G1$ and $G2$ are universally correct then so is F.

Proof See Appendix D. □

The reader should not be alarmed by the complexity of the expressions defining the observable sorts in the verification conditions of the above theorem. First, in practice the signature morphisms do not result in non-trivial renaming at the level of external names, which are the ones the user has to deal with. Second, the ML typechecker may easily be modified to compute them mechanically.

5.4 Modular decomposition: the general case

The three special cases presented in the preceding subsections are intended to provide a clear illustration of the way that the definition of a functor by modular decomposition should be verified. We will not attempt here to formulate precisely an appropriate general theorem, since this would require richer technical apparatus. We believe that the development of appropriate notation and terminology which would allow such a general verification condition to be expressed in a precise and understandable form, in the presence of non-trivial sharing requirements on module interfaces, is an important research task.

Very roughly, the definition of a functor by modular decomposition gives rise to a finite directed acyclic graph with one maximal node, where the graph nodes are labelled with Extended ML signatures and the graph edges are of two distinct kinds. First, there are edges corresponding to applications of functors used in the decomposition; then the source node (resp. target node) of the edge is labelled with the input (resp. output) signature of the functor. Second, there are edges between nodes labelled with a functor output signature and nodes labelled with a functor input signature, which correspond to matching the output signature against the input signature. Then, we add to this graph a node labelled with the input signature of the decomposed functor and edges matching it against all the (input) signatures in the minimal nodes of the decomposition graph, and another node labelled with the output signature of the decomposed functor, with an edge matching the (output) signature in the maximal node of the graph against it.

As in the examples above, such a decomposition graph determines a diagram in the category of algebraic signatures. The nodes of the decomposition graph induce nodes of the diagram labelled with the algebraic signatures determined by the Extended ML signatures labelling the graph nodes. The signature matching edges in the graph induce corresponding edges (but going into the opposite direction) in the diagram labelled by algebraic signature morphisms determined by the ML matching rules. Each functor application edge in the graph decomposes into a pair of algebraic signature inclusions with a common source expressing the sharing declaration in the functor heading, and with the functor input and output signatures, respectively, as their targets.

On the resulting diagram we superimpose pairs of morphisms with a common source expressing the sharing requirements present in the decomposition (like the signature $\Sigma 0_{com}$ with morphisms $\iota 01_{com} : \Sigma 0_{com} \to \Sigma 01_{in}$ and $\iota 02_{com} : \Sigma 0_{com} \to \Sigma 02_{in}$ in Subsection 5.3). The required sharing between the input and output signatures of the decomposed functor is included in the same way. Then, we have to determine algebraic signature morphisms from the algebraic signatures expressing the sharing requirements such that the commutativity of the resulting diagram ensures that the sharing requirements are satisfied via structures arising in the functor body as a result of functor application (like the signature morphisms $\rho 1_{inter} : \Sigma 0_{com} \to \Sigma 1_{shr}$ and $\rho 2_{inter} : \Sigma 0_{com} \to \Sigma 2_{shr}$ in Subsection 5.3, which guarantee the sharing represented by the signature $\Sigma 0_{com}$). An additional complication is that there may be more than one way to ensure that some sharing requirement is satisfied, and so we have to allow some algebraic signatures expressing sharing requirements to be decomposed (as the algebraic signature Σ_{shr} was decomposed into Σ'_{shr} and Σ''_{shr} in Subsection 5.3).

Although this construction seems complicated, it is mechanisable and so would be carried out by computer-based support tools. In fact, most of this construction is implicitly performed by the Standard ML typechecker already.

Finally, assuming that the definition of the functor is correct according to the ML typechecking rules, which determines the algebraic signature diagram sketched above, we can check the following verification condition: for each edge in the decomposition graph which matches a signature $SIG1$ (the signature labelling the source of the edge) against $SIG2$ (the signature labelling the target of the edge), $SIG1$ must entail $SIG2$ via the algebraic signature morphisms determined by the ML matching rules up to behavioural equivalence w.r.t. a set of observable sorts consisting of all the sorts in the algebraic signature of $SIG2$ corresponding to sorts in the algebraic signatures expressing sharing requirements (via the morphisms in the algebraic signature diagram going into the corresponding diagram node determined as sketched above). This ensures that whenever the functors used in the decomposition are universally correct, so is the decomposed functor.

The results in [Sch 86] concerning modular decomposition are weaker than the general result sketched above and Theorems 5.1, 5.2 and 5.5 in that he requires interfaces to match exactly, except that the actual inputs and outputs are permitted to be larger than required by the corresponding interfaces. In particular, an actual input or output is not permitted to share more than required, and (much more significantly) interfaces must match "literally" rather than only up to behavioural equivalence as we require (this is the upshot of condition (b) of Definition 3.2.10 in [Sch 86]). We have given verification conditions which seem to be as weak as possible under the constraint of being expressible as "local" signature matching requirements, while still guaranteeing the correctness of the decomposition.

In this section we have discussed conditions under which functors can be correctly implemented by decomposition into simpler functors. These functors can then themselves be implemented using the same technique of modular decomposition or by supplying an "abstract program" (see Section 6).

Of course, we would not expect the formal development of realistic programs to proceed in practice without backtracking, mistakes and iteration, and we do not claim to remove the possibility of unwise design decisions. One problem is that it is often very difficult to get interface specifications right the first time and so for example when implementing a functor by decomposition into simpler functors it may well be necessary to adjust the interfaces both in order to obtain a decomposition which is correct according to the theorems above and to resolve problems which arise later while implementing the simpler functors. If a decomposition has been proved correct then some changes to the interfaces may be made without affecting correctness: for example, in any of the simpler functors the output interface may be strengthened or the input interface weakened without problems (provided the required sharing between input and output is preserved). It is also possible to modify the interfaces of the functor being decomposed by weakening its output signature or strengthening its input signature. This will preserve the correctness of the decomposition but since it changes the specification of the functor such changes must be cleared with the functor's clients (higher-level functors which use it and/or the customer). Once we have made such a change to an interface we can also change interfaces it is required to match in order to take advantage of the modification. Then, provided we are able to prove that the syntactic and semantic correctness conditions referring to these interfaces hold, overall correctness is still assured since the remaining interfaces are unaffected.

Functors correspond to (parameterised) abstract data types. We are free to change the implementation (body) of a functor at any time. As long as the new implementation is universally correct with respect to the functor heading, this change is invisible to the rest of the program. This is ensured since explicit interfaces insulate a functor implementation from its use.

6 System design: refinement of abstract programs

The previous section discussed conditions under which functors can be correctly implemented by decomposition into simpler functors. At some point it is necessary to actually write code to implement a functor. In this section we discuss how correct code can be developed gradually by means of stepwise refinement of loose abstract programs (Extended ML structures containing a mixture of Standard ML function and type definitions and non-executable axioms). Our goal is to arrive at a functor body containing only executable code which is universally correct with respect to the given functor heading.

6.1 Simple correctness and stability

Although the notion of universal correctness expresses the correctness property one should aim at in program development, it is very inconvenient as a basis for verification of abstract programs as pointed out in [Sch 86]. There are at least two unexpected problems. First of all, we are not allowed to rely on the input specification literally, but only on its observable consequences. Second, we are required to consider all possible structures to which the functor may be applied rather than considering just structures over the input signature.

A solution presented in [Sch 86] is to split universal correctness into three properties which will be ensured separately:

Definition 6.1 Consider an Extended ML functor of the form:

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

1. F is *simply correct* if for any Σ_{in}-algebra $A \in Mod[SIG_{in}]$, $\mathcal{F}_{bsem}(A) \models^{sorts(\Sigma_{shr})}_{\Sigma_{out}} SIG_{out}$.

2. F is *simply consistent* if $Mod[SIG_{in}] \subseteq Dom(F)$.

3. F is *stable* if for any algebraic signature Σ_{arg} and fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$, any Σ_{arg}-algebras A, B such that $A \equiv_{sorts(\texttt{Perv})} B$, and for any $A' \in \mathcal{F}_{gres}(A[\sigma])$ there exists $B' \in \mathcal{F}_{gres}(B[\sigma])$ such that $A' \equiv_{sorts(\texttt{Perv})} B'$ (recall that all Extended ML signatures implicitly contain `Perv`).

The main idea behind the definition of stability is that a functor is stable if and only if it preserves behavioural equivalence. The apparent asymmetry whereby the choice of B' depends on the choice of A' is unimportant since the preconditions are symmetric.

Theorem 6.2 *An Extended ML functor is universally correct whenever it is simply correct, simply consistent and stable.*

Proof Consider an Extended ML functor of the form

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

which is simply correct, simply consistent and stable. Consider any algebraic (argument) signature Σ_{arg}, fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$, and any Σ_{arg}-algebra A such that $A \models^{sorts(\texttt{Perv})}$ **translate** SIG_{in} **by** σ. Let B be any Σ_{arg}-algebra such that $A \equiv_{sorts(\texttt{Perv})} B$ and $B|_\sigma \in Mod[SIG_{in}]$ (such a B exists by the definition of behavioural satisfaction). By the simple consistency of F, $B|_\sigma \in Dom(F)$. That is, $\mathcal{F}_{gres}(B[\sigma]) \neq \emptyset$, hence by the stability of F (with A and B interchanged), $\mathcal{F}_{gres}(A[\sigma]) \neq \emptyset$, i.e. $A|_\sigma \in Dom(F)$.

Then, consider an arbitrary $A' \in \mathcal{F}_{gres}(A[\sigma])$. By the stability of F, there exists $B' \in \mathcal{F}_{gres}(B[\sigma])$ such that $A' \equiv_{sorts(\texttt{Perv})} B'$. By the definition of $\mathcal{F}_{gres}$, $B'|_{F[\sigma]} \in \mathcal{F}_{bsem}(B|_\sigma)$. Hence, by the simple correctness of F, $B'|_{F[\sigma]} \models^{sorts(\Sigma_{shr})} SIG_{out}$. By the definition of behavioural satisfaction, there exists a Σ_{out}-algebra C such that $C \equiv_{sorts(\Sigma_{shr})} B'|_{F[\sigma]}$.

Now, consider the unique $F(\Sigma_{arg}[\sigma])$-algebra $\widehat{B}$ such that $\widehat{B}|_{\iota'_{out}} = B$ and $\widehat{B}|_{F[\sigma]} = C$. The existence and uniqueness of $\widehat{B}$ is ensured by the construction of $F(\Sigma_{arg}[\sigma])$ and the fact that $(B'|_{F[\sigma]})|_{\Sigma_{shr}} = C|_{\Sigma_{shr}}$. By Lemma A.1, $\widehat{B} \equiv_{sorts(\texttt{Perv})} B'$ and so $\widehat{B} \equiv_{sorts(\texttt{Perv})} A'$. This also proves that $A' \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $F[\sigma]$. □

Simple correctness is a property which can be verified "statically" in the sense that we do not have to consider all the different ways in which the functor can be applied. It is enough to consider only structures over the input signature. Moreover, while verifying simple correctness we are allowed to pretend that the input structure satisfies the input signature literally. This is therefore a condition which we will expect a user of our methodology to verify for each of the functors he defines.

Stability is a different matter. It is not reasonable to expect a user to verify the stability of his functors one by one. This property should be guaranteed by the designer of the programming language used. Any language which is designed to support data abstraction should ensure that only stable functors (modules, packages, clusters, etc.) are definable. See [Sch 86] for a much more complete discussion of this issue.

Working hypothesis Every functor definable in Standard ML is stable.

Discussion We could turn this working hypothesis into a theorem for the purely functional subset of Standard ML we are using here, under the type discipline described in Section 2. The proof would be based on a formal algebraic semantics of this language and would involve a lot of tedious work. To prove the corresponding theorem, or even state it precisely, for full Standard ML would require developing an integrated algebraic view of (at least) exceptions, polymorphism, higher-order functions, imperative features, partial functions and non-terminating functions. This is an important long-term goal which we are confident may be achieved, but it is orthogonal to the issues discussed in this paper. □

Under the above hypothesis, any simply correct functor whose body is coded in Standard ML is universally correct (recall that every Standard ML functor is defined for all structures over its algebraic input signature, and so is obviously simply consistent). However, this is not guaranteed for Extended ML functors in general, and it would not be reasonable to expect this of any specification language. The power and flexibility of algebraic specification languages are in fundamental conflict with the requirement of stability. Extended ML functors arising during the development process need not be universally correct; our methodology guarantees only that they are simply correct by requiring refinement steps to preserve this property. Consequently, when we arrive at a Standard ML functor, which is always our goal, it will be simply correct and simply consistent, and it will be stable by the above working hypothesis, and hence by Theorem 6.2 it will be universally correct.

One might argue that simple consistency is a requirement which should be imposed on every Extended ML functor which arises in the program development process. This would seem to prevent blind alleys in program development. But since even a total functor may have no computable (or acceptably efficient) realisation, we cannot hope to avoid blind alleys in general anyway. It might be advisable to check for simple consistency at each stage of development but this is not required for correctness and is not a part of our methodology.

The concepts of universal correctness and stability are a bit different from the corresponding ones in [Sch 86]. If we were to exactly translate his definitions to the context of Extended ML we would have to require the fitting morphism σ to be injective. Thus, it may seem that our notions of universal correctness and stability are more restrictive than his. But as we accept the above view that stability is to be ensured by the programming language in use, the two notions of stability coincide since for any given application of a functor to a structure we can add sharing constraints to the input signature of the functor so as to make the fitting morphism injective without affecting the correctness of the code in the functor body.

6.2 Abstract programs

The conclusion of the discussion in the previous subsection is that the user's only obligation is to produce code for the functor body in such a way that the resulting functor definition is simply correct. The user may begin by writing a loose abstract program containing a mixture of axioms and executable code and then gradually refine this in a stepwise manner until a version containing only Standard ML code is obtained.

The following theorem gives the condition which the first version (and in fact all versions) of the body must satisfy in order to ensure simple correctness of the functor.

Theorem 6.3 *An Extended ML functor of the form*

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

is simply correct if and only if

$$(\textbf{translate } (\textbf{derive from } SIG_{in} \textbf{ by } \iota_{si}) \textbf{ by } \iota_{sb}) \cup BODY \models_{\Sigma_{out}}^{sorts(\iota_{out}(\Sigma_{shr}))} SIG_{out}$$

where $\iota_{si} : \Sigma_{in} \cap \Sigma_{body} \hookrightarrow \Sigma_{in}$ *and* $\iota_{sb} : \Sigma_{in} \cap \Sigma_{body} \hookrightarrow \Sigma_{body}$ *are the algebraic signature inclusions.*[12]

Proof Directly from the definition of the basic semantics of Extended ML functors. □

We could employ this theorem to check the simple correctness of each version of the functor body obtained as a result of successive refinement steps. But in practice this is inconvenient since subsequent versions of the body will become increasingly more detailed and lower level, making it difficult to relate them in a simple way to the output interface. It is much more natural to relate each new version of the functor body directly with the previous one. Then we can exploit the simple correctness of the previous version to establish the simple correctness of the new version as follows:

Corollary 6.4 *If an Extended ML functor of the form*

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY$$

is simply correct and

$$(\textbf{translate } (\textbf{derive from } SIG_{in} \textbf{ by } \iota'_{si}) \textbf{ by } \iota'_{sb}) \cup BODY' \models_{\Sigma_{body}}^{sorts(\iota_{sb}(\Sigma_{in} \cap \Sigma_{body}))} BODY$$

where $\iota'_{si} : \Sigma_{in} \cap \Sigma_{body'} \hookrightarrow \Sigma_{in}$ *and* $\iota'_{sb} : \Sigma_{in} \cap \Sigma_{body'} \hookrightarrow \Sigma_{body'}$*, then*

$$\texttt{functor } F(X : SIG_{in}) : SIG_{out} \texttt{ sharing } \textit{sharing-decl} = BODY'$$

is simply correct as well.

Proof Trivial, by Theorem 6.3 and the definition of **translate**. □

The conditions required in the above theorem and its corollary may be established using the results given in Section 3.

[12]The horrible expression on the left-hand side of the entailment should be thought of as $SIG_{in} \cup BODY$ (and similarly for Corollary 6.4). Since all the morphisms here are unambiguously determined by the context, we will use this simplified form in presenting the verification of coding steps in Section 7.

The refinement process sketched above is reminiscent of the use of *abstractor implementations* as discussed in [ST 88b]. The condition in the above corollary may be rephrased as the requirement that $BODY \overset{\alpha}{\underset{\kappa}{\rightsquigarrow}} BODY'$ (in the context of SIG_{in}) where α is an abstractor corresponding to the behavioural equivalence involved and κ is a **derive** step which forgets any new auxiliary types and operations introduced by $BODY'$. The technicalities concerning the use, composition, etc. of abstractor implementations developed in [ST 88b] may be used directly here. Recall that abstractor implementations correspond to "tailor-made" implementations which are specially developed to fit into some particular context. For example, suppose that $BODY = BODY_1; BODY_2$. Then $BODY$ may be refined to $BODY' = BODY_1'; BODY_2$ where $BODY_1'$ is a refinement of $BODY_1$. If $BODY_2$ is already Standard ML code, then when refining $BODY_1$ we can take advantage of our knowledge of $BODY_2$ which is the only context in which the final realisation of $BODY_1$ will be used. For correctness of the refinement it is sufficient to require that $BODY_1'$ entails $BODY_1$ up to the equivalence which results when $\equiv_{sorts(\iota_{sb}(\Sigma_{in}\cap\Sigma_{body}))}$ is "pushed through" $BODY_2$. See [ST 88b] for details; we only note here that the resulting equivalence might not be a behavioural equivalence with respect to any set of sorts, and so the results of Section 3 may not apply directly. This represents one extreme in the tradeoff between making the verification conditions as weak as possible and making them uniform enough so that general proof methods are easily applicable.

Instead of refining the functor body, we could refine the output signature of the functor. At some point this would result in executable code which could then be used as the functor body. But this is inappropriate for two reasons. First of all, it seems important to clearly separate specification from realisation. Second, refinement of the body and refinement of the output signature have different consequences for the way the functor may be used. Refinement of the body is a local decision visible to the programmer who implements the functor and invisible to clients who may want to use the functor. This allows later changes of representation, etc. In contrast, refinement of the output signature is a higher-level design decision which is intended to be exploited by clients.

6.3 Hierarchically structured abstract programs

The previous subsection only treated the special case of "flat" abstract programs, i.e. abstract programs not containing substructures. Substructures provide a way to structure functor bodies into conceptual units, in additional to the means already provided by functor decomposition. Structuring functor bodies in this way also gives a corresponding structure to the verification process.

Hitherto we have strictly adhered to a regime of insulating system units from their clients by means of interfaces (Extended ML signatures). Now, the units of interest are substructures of functor bodies and their clients are the functor bodies themselves. Syntactically, this naturally leads to the requirement that Extended ML substructure declarations always explicitly include the Extended ML signature which the substructure is supposed to fit. Just as before, we view this signature as containing all the information available about the substructure. Consequently, Extended ML substructures can be seen as abstractions in the sense of [MacQ 86].

This view of substructures means that we can view them as (calls of) locally-defined parameterless functors. The verification conditions are thus very much reminiscent of those we stated for functor decomposition. The only difference is that substructures implicitly import the part of the functor body which precedes the substructure declaration. There is no interface at this point insulating the substructure from the details of the preceding code.

Thus, for an Extended ML functor declaration of the form

```
functor F(X : SIG_in) : SIG_out sharing sharing-decl
  = struct
        part1
        structure A : SIG sharing sharing-declA = STR
        part2
    end
```

we have to verify that:

- The structure STR together with $part1$ entails the signature SIG up to behavioural equivalence.
- The structure consisting of $part1$, $part2$ and the substructure declaration

  ```
  structure A : SIG
  ```

  entails SIG_{out} up to behavioural equivalence.

If SIG_{out} contains a substructure A with signature SIG_{out}^{A}, then SIG_{out}^{A} must be a behavioural consequence of SIG.

There is a sense in which the first of these conditions amounts to establishing a lemma which is then used in checking the correctness of the functor body as a whole in the second condition. Moreover, this lemma is the only information available about the substructure A while doing this check.

It should be clear that the above functor F may be rewritten as follows:

```
functor F(X : SIG_in) : SIG_out sharing sharing-decl
  = let structure B1 = Part1(X) in
    Part2(X, B1, SubA(X, B1)) end
```

where $Part1$, $SubA$ and $Part2$ are functors defined as follows:

```
functor Part1(X : SIG_in) : PART1 sharing sharing-decl1
  = struct part1 end

functor SubA(X : SIG_in, B1 : PART1 sharing sharing-decl1) : SIG sharing sharing-declA
  = local  open B1
    in     STR
    end
```

and

```
functor Part2(X : SIG_in, B1 : PART1, A : SIG sharing sharing-decl1 ∪ sharing-declA)
            : SIG_out sharing sharing-decl
  = struct
       open B1
       structure A = A
       part2
    end
```

In the above, $PART1$ denotes an Extended ML signature corresponding exactly to the structure formed from $part1$, and $sharing\text{-}decl1$ describes all of the sharing which arises by construction between types and values in $part1$ and the input structure X. The names in the sharing declarations must be adjusted appropriately to make these functor declarations well-formed; for example, references in $sharing\text{-}declA$ to names in $part1$ must be converted into corresponding references to $B1$.

In the above decomposition of F, all the interfaces fit exactly and so by the results of Section 5 the universal correctness of F is ensured if the functors $Part1$, $SubA$ and $Part2$ are universally correct. As in Subsection 6.1, universal correctness of these functors is achieved when they are stable, simply correct and simply consistent. Under the working hypothesis in Subsection 6.1, and recalling that Standard ML functors are always simply consistent, we can concentrate on simple correctness. By Theorem 6.3:

- $Part1$ is (trivially) simply correct;
- $SubA$ is simply correct if $SIG_{in} \cup PART1 \cup STR \models^{sorts(\Sigma_{shr}^{SubA})} SIG$
- $Part2$ is simply correct if $SIG_{in} \cup PART1 \cup SIG \cup part2 \models^{sorts(\Sigma_{shr}^{F})} SIG_{out}$

In the last two conditions we have omitted the ugly but formally necessary translations of specifications to the union signature determined by the ML typechecking rules. The exact formulation would follow the pattern of Theorem 6.3. The algebraic signatures Σ_{shr}^{SubA} and Σ_{shr}^{F} embody the sharing between the input and output signatures of functors $SubA$ and F respectively.

These considerations result in the following corollary:

Corollary 6.5 *Consider an Extended ML functor F as above together with its decomposition into functors $Part1$, $SubA$ and $Part2$. Suppose that the following verification conditions hold:*

1. $SIG_{in} \cup PART1 \cup STR \models^{sorts(\Sigma_{shr}^{A})} SIG$
2. $SIG_{in} \cup PART1 \cup SIG \cup part2 \models^{sorts(\Sigma_{shr}^{F})} SIG_{out}$

Then, if $Part1$, $SubA$ and $Part2$ are stable and simply consistent then F is universally correct. □

The assumption of simple consistency for $SubA$ and $Part2$ hides the requirement that the original split of the functor body was sensible in the sense that the axioms in STR and $part2$ do not further constrain types and values introduced already in $part1$, and likewise for $part2$ w.r.t. the types/values of STR.

It is perhaps surprising that conditions 1 and 2 of the above corollary do not by themselves guarantee the simple correctness of F. This is because a non-stable $part2$ may take advantage of non-observable consequences of SIG. This does not cause problems since stepwise refinement using an appropriate modification of the verification condition in Corollary 6.4 preserves conditions 1 and 2 and ultimately results in stable code. This means that we can start with loose versions of $part1$, STR and $part2$ which satisfy conditions 1 and 2 (but which may not be stable) and then refine them, independently if desired, to eventually produce Standard ML code which will yield a universally correct definition for F.

Structure STR may be an arbitrary structure expression, not necessarily an abstract program. So we can decompose STR into simpler functors just as in Section 5. Then condition 1 of Corollary 6.5 should be replaced by the appropriate verification condition for the decomposition, with SIG as the output signature for the decomposition and $SIG_{in} \cup PART1$ as the input signature.

We have chosen to view substructures as abstractions, which means that the substructure body is insulated from the rest of the functor body by its interface. It is possible to consider ordinary non-abstract structures instead. Then substructures serve only to package collections of type and value definitions. An explicitly-declared interface may then be useful to summarise the properties of these types and values, but it is not an absolute barrier between the substructure and the rest of the functor body. An advantage of using abstract substructures is that we may come back later and choose another implementation; as long as the interface is unchanged the program will still work.

7 An example

In this section the development of a complete Standard ML program from a high-level Extended ML specification is exhibited. To keep the proofs simple we will assume that all functions are total (this is indeed the case in the resulting program).

Informal specification An inventory control system for a warehouse is required. This should keep track of the number currently in stock of each item in the warehouse. Items are taken out of the warehouse one at a time but they may be brought to the warehouse in larger batches. Certain items in the warehouse are usable as replacements for other items which may be temporarily out of stock. In case some item is currently out of stock the system should be able to locate an appropriate replacement item which is in stock.

There is a fixed (but arbitrary) collection of different items which may be stored in the warehouse. The decision as to which of these items may be replaced by which other items is also fixed but arbitrary.

Step 0

The initial formal specification of the required system is given by the following Extended ML functor heading:

```
functor Warehouse(I:ITEM):WAREHOUSE sharing Item = I
```

where `ITEM` and `WAREHOUSE` are Extended ML signatures as follows:

```
signature ITEM =
   sig eqtype item
       val replaces : item * item -> bool
       axiom replaces(i,i)
   end

signature WAREHOUSE =
   sig structure Item : ITEM
       type warehouse
       val empty : warehouse
       val put : Item.item * nat * warehouse -> warehouse
       val amount : Item.item * warehouse -> nat
       val exists_replacement: Item.item * warehouse -> bool
       val find_replacement : Item.item * warehouse -> Item.item
       val take : Item.item * warehouse -> warehouse
```

```
    axiom put(i,0,w) = w
    axiom put(i,m,put(j,n,w)) = put(j,n,put(i,m,w))
    axiom put(i,m,put(j,n,w)) = put(i,n+m,w)  if i=j
    axiom amount(i,empty) = 0
    axiom amount(i,put(j,n,w)) = amount(i,w)+n  if i=j
                               = amount(i,w)    if i<>j
    axiom exists_replacement(i,w) <=>
                 exists j. (Item.replaces(j,i) & amount(j,w)>0)
    axiom exists_replacement(i,w) =>
                 (Item.replaces(find_replacement(i,w),i)
                  & amount(find_replacement(i,w))>0)
    axiom amount(i,w)>0 => find_replacement(i,w) = i
    axiom take(i,empty) = empty
    axiom take(i,put(j,n,w)) = put(j,n-1,w)        if i=j & n>0
                             = put(j,n,take(i,w))  if i<>j
end
```

In this specification, values of type `warehouse` represent states of the warehouse. The empty warehouse is represented by `empty`, and the functions `put` and `take` update the state of the warehouse by adding more of an item and by removing one of an item. The functions `amount`, `exists_replacement` and `find-replacement` may be used for querying the current state of the warehouse. The set of items which may be stored in the warehouse is taken to be a parameter of the system along with the `replaces` relation. By using the Standard ML declaration `eqtype item` rather than `type item` in the signature `ITEM`, we require that `item` *admits equality*, i.e. that it comes equipped with the equality function `=:item*item->bool` which can be used in code as well as axioms.

Step 1

Design decision (decomposition) We implement `put` using a function `putone` which adds just a single item to the warehouse. Exactly how `put` is expressed using `putone` is left open for now.

We need two functors:

```
Warehouse'(I:ITEM):WAREHOUSE' sharing Item = I
Put(W:WAREHOUSE'):WAREHOUSE sharing Item = W.Item
```

where `WAREHOUSE'` is just like `WAREHOUSE` except with

```
putone : Item.item * warehouse -> warehouse
```

in place of

```
put : Item.item * nat * warehouse -> warehouse
```

and axioms involving `put` replaced by axioms involving `putone`, viz:

```
axiom putone(i,putone(j,w)) = putone(j,putone(i,w))
axiom amount(i,empty) = 0
```

```
axiom amount(i,putone(j,w)) = amount(i,w)+1  if i=j
                            = amount(i,w)    if i<>j
axiom take(i,empty) = empty
axiom take(i,putone(j,w)) = w                     if i=j
                          = putone(j,take(i,w))  if i<>j
```

where the axioms for `exists_replacement` and `find_replacement` are just as before.

Then we can implement `Warehouse` in terms of these functors as follows:

```
functor Warehouse(I:ITEM):WAREHOUSE sharing Item = I
   = Put(Warehouse'(I))
```

Verification Typechecks okay. All interfaces match exactly so conditions 1-3 of Theorem 5.2 are satisfied as a consequence of Proposition 3.4. □

Step 2

Design decision (coding) Implement the functor `Put` by coding `put` using `putone` in the obvious way.

```
functor Put(W:WAREHOUSE'):WAREHOUSE sharing Item = W.Item
   = struct structure Item : ITEM = W.Item
            open W
            fun put(i,0,w) = w
              | put(i,n+1,w) = put(i,n,putone(i,w))
     end
```

Verification Typechecks okay. According to Theorem 6.3, we have to show that

$$\mathtt{WAREHOUSE}' \cup \mathit{body} \models^{\mathit{sorts}(\mathtt{ITEM})}_{\mathit{Sig}(\mathtt{WAREHOUSE})} \mathtt{WAREHOUSE}$$

where *body* is the body of `Put`. This follows by Proposition 3.4 since we can show by induction on the natural numbers that the axioms in `WAREHOUSE` involving `put` hold, and the rest hold trivially. □

Step 3

Design decision (decomposition) Decompose `Warehouse'` into three functors. The first functor provides bags (multisets) of items which will be used to represent the contents of the warehouse, the second functor handles warehouse queries (`amount`, `exists_replacement` and `find_replacement`), and the third functor combines these to implement `Warehouse'`.

We need three functors:

```
Bag(I:SMALLITEM):BAG sharing Item = I
Queries(B:SMALLBAG,I:ITEM sharing B.Item.item = I.item)
      :QUERIES sharing Item = I and warehouse = B.bag and amount = B.count
Combine(B:BAG,R:QUERIES sharing B.Item.item = R.Item.item
                        and B.bag = R.warehouse and B.count = R.amount)
      :WAREHOUSE' sharing Item = R.Item
```

where SMALLITEM, BAG, SMALLBAG and QUERIES are as follows:

```
signature SMALLITEM =
   sig eqtype item
   end

signature BAG =
   sig structure Item : SMALLITEM
       type bag
       val empty : bag
       val add : Item.item * bag -> bag
       val count : Item.item * bag -> nat
       val remove : Item.item * bag -> bag
       axiom add(i,add(j,b)) = add(j,add(i,b))
       axiom count(i,empty) = 0
       axiom count(i,add(j,b)) = count(i,b)+1  if i=j
                               = count(i,b)    if i<>j
       axiom remove(i,b) = b  if count(i,b)=0
       axiom remove(i,add(j,b)) = b                    if i=j
                                = add(j,remove(i,b))  if i<>j
   end

signature SMALLBAG =
   sig structure Item : SMALLITEM
       type bag
       val count : Item.item * bag -> nat
   end

signature QUERIES =
   sig structure Item : ITEM
       type warehouse
       val amount : Item.item * warehouse -> nat
       val exists_replacement : Item.item * warehouse -> bool
       val find_replacement : Item.item * warehouse -> Item.item
       axiom exists_replacement(i,w) <=>
                        exists j. (Item.replaces(j,i) & amount(j,w)>0)
       axiom exists_replacement(i,w) =>
                        (Item.replaces(find_replacement(i,w),i)
                         & amount(find_replacement(i,w),w)>0)
       axiom amount(i,w)>0 => find_replacement(i,w)=i
   end
```

Then we can implement Warehouse' in terms of these functors as follows:

```
functor Warehouse'(I:ITEM):WAREHOUSE' sharing Item = I
   = let structure B = Bag(I) in
     Combine(B,Queries(B,I)) end
```

Verification Typechecks okay. All interfaces fit exactly except that `I` is used by `Bag` as a `SMALLITEM` so we have to show that `ITEM` fits `SMALLITEM` and `B:BAG` is used by `Queries` as a `SMALLBAG` so we have to show that `BAG` fits `SMALLBAG` (both obvious). □

Step 4

Design decision (coding) Implement `Combine` by changing the names of the values in the input structure `B` appropriately.

```
functor Combine(B:BAG,R:QUERIES sharing B.Item.item = R.Item.item
                            and B.bag = R.warehouse and B.count = R.amount)
               :WAREHOUSE' sharing Item = R.Item
   = struct open R
            val empty = B.empty
            val putone = B.add
            val take = B.remove
     end
```

Verification Typechecks okay. Note that `open R` turns all the components of `R`, including the substructure `Item:ITEM`, into parts of the body. It is easy to verify that the axioms of `WAREHOUSE'` are implied by the axioms of `BAG`, `QUERIES` and the body of `Combine`. □

Step 5

Design decision (coding) Implement `Queries` using a function `find`, which finds a suitable replacement if there is one, to implement `exists_replacement` and `find_replacement`.

```
functor Queries(B:SMALLBAG,I:ITEM sharing B.Item.item = I.item)
           :QUERIES sharing Item = I and warehouse = B.bag and amount = B.count
   = struct structure Item : ITEM = I
            type warehouse = B.bag
            val amount = B.count
            val find : Item.item * warehouse -> Item.item
            axiom Item.replaces(find(i,w),i)
            axiom (exists j. (Item.replaces(j,i) & amount(j,w)>0)) =>
                        amount(find(i,w),w)>0
            fun exists_replacement(i,w) = amount(find(i,w),w)>0
            fun find_replacement(i,w) = if amount(i,w)>0 then i
                                        else find(i,w)
     end
```

Verification Typechecks okay. According to Theorem 6.3 (and the definition of correctness for multi-argument functors), we have to show that

$$\mathtt{SMALLBAG} \cup \mathtt{ITEM} \cup \mathit{body} \models^{\mathit{sorts}(\mathtt{SMALLBAG}) \cup \mathit{sorts}(\mathtt{ITEM})}_{\mathit{Sig}(\mathtt{QUERIES})} \mathtt{QUERIES}$$

where *body* is the body of `Queries`. This follows by Proposition 3.4. □

OOPS!

One way to implement `find` involves stepping through the domain of items until an allowable replacement is found which is in stock. But this requires adding a function to `ITEM`, which changes the original specification. The customer agrees to this change. We can add this now and continue the development after checking that the steps taken so far are still valid.

Redeclare `ITEM` to be just like the previous `ITEM` except with the addition of

```
val next : item -> item
val iternext : nat * item -> item
axiom iternext(0,i) = i
axiom iternext(n+1,i) = iternext(n,next(i))
axiom forall i,j. exists n. (n>0 & iternext(n,i)=j)
```

Here, `iternext` is a hidden value which is handy for specifying `next`.

Verification Since the new version of `ITEM` is stronger than the old version, the only verification conditions from the previous steps we have to check are those of the form $\ldots \models \ldots$`ITEM`$\ldots$. In each case, the specification on the left-hand side of the entailment contains a "matching" occurrence of `ITEM`, so this reduces to `ITEM` $\models$ `ITEM`. □

Step 6

Design decision (refinement) Refine `Queries` by coding `find` as a search through the domain of items using the function `next`.

```
functor Queries(B:SMALLBAG,I:ITEM sharing B.Item.item = I.item)
          :QUERIES sharing Item = I and warehouse = B.bag and amount = B.count
   = struct structure Item : ITEM = I
            type warehouse = B.bag
            val amount is B.count
            fun find'(i,j,w) = if i=j then i
                                   else if amount(j,w)>0 andalso Item.replaces(j,i)
                                        then j
                                        else find'(i,Item.next(j),w)
            fun find(i,w) = find'(i,Item.next(i),w)
            fun exists_replacement(i,w) = amount(find(i,w),w)>0
            fun find_replacement(i,w) = if amount(i,w)>0 then i
                                            else find(i,w)
     end
```

Verification Typechecks okay. It is routine but messy to prove that the code for `find` (together with the axioms in `ITEM`) entails the axioms in the previous version of `Queries`. Thus by Proposition 3.4 we can apply Corollary 6.4. □

Step 7

Design decision (coding) Implement `Bag` using ML's datatype definition facility. (In practice, common types such as this one would have one or more pre-verified implementations in the library, and so this step would consist of selecting one and ensuring that the interfaces match. This might require renaming input and result types and values.)

```
functor Bag(I:SMALLITEM):BAG sharing Item = I
   = struct structure Item : SMALLITEM = I
            datatype bag = empty | add of Item.item * bag
            fun count(i,empty) = 0
              | count(i,add(j,b)) = if i=j then count(i,b)+1
                                    else count(i,b)
            fun remove(i,empty) = empty
              | remove(i,add(j,b)) = if i=j then b
                                     else add(j,remove(i,b))
     end
```

Verification Typechecks okay. According to Theorem 6.3, we have to show

$$\texttt{SMALLITEM} \cup \mathit{body} \models^{\mathit{sorts}(\texttt{SMALLITEM})}_{\mathit{Sig}(\texttt{BAG})} \texttt{BAG}$$

where *body* is the body of `Bag`.

We can apply Corollary 3.7 to split the axioms of `BAG` into a set which follow directly from `SMALLITEM` $\cup$ *body* (i.e. all but the first one) and a set whose observable consequences must be shown to follow from `SMALLITEM` $\cup$ *body* (the first axiom).

The observable consequences of the first axiom

$$\texttt{add}(i, \texttt{add}(j, b)) = \texttt{add}(j, \texttt{add}(i, b))$$

are all the equations of the form

$$\texttt{count}(k, \Gamma(\texttt{add}(i, \texttt{add}(j, t)))) = \texttt{count}(k, \Gamma(\texttt{add}(j, \texttt{add}(i, t))))$$

where Γ is a context of sort `bag` yielding a result of sort `bag`, and t is a term of sort `bag` containing variables of sort `Item.item` only.

Lemma *From* `SMALLITEM` $\cup$ *body it follows that for any* $b, b' : \texttt{bag}$ *such that* $\texttt{count}(i, b) = \texttt{count}(i, b')$ *for all* i*:*

(a) $\texttt{count}(i, \texttt{remove}(j, b)) = \texttt{count}(i, \texttt{remove}(j, b'))$

(b) $\texttt{count}(i, \texttt{add}(j, b)) = \texttt{count}(i, \texttt{add}(j, b'))$

(c) $\texttt{count}(i, \texttt{add}(j, \texttt{add}(k, b))) = \texttt{count}(i, \texttt{add}(k, \texttt{add}(j, b')))$

for all i, j *and* k*.*

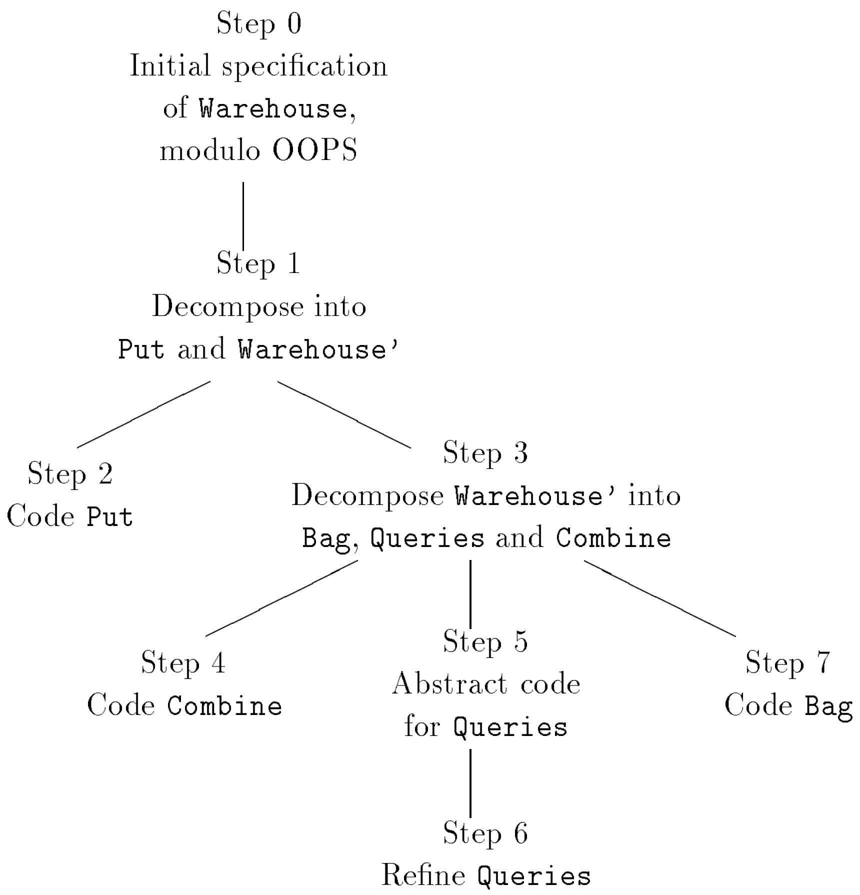
Proof By cases on $i{=}j$, after establishing for (a) that

$$\begin{array}{lcll} \texttt{count}(i, \texttt{remove}(j, b'')) & = & \texttt{count}(x, b'') - 1 & \text{if } i{=}j \text{ and } \texttt{count}(x, b'') > 0 \\ & = & \texttt{count}(x, b'') & \text{otherwise} \end{array}$$

for any b''. □

Using this lemma it is easy to show by induction on the structure of the context Γ that the desired equations follow from $\texttt{SMALLITEM} \cup \mathit{body}$. □

All functor bodies are now expressed entirely in Standard ML, so we are finished. The functors appearing in the final program are given above under steps 1, 2, 3, 4, 6 and 7. The following tree shows the dependencies between the development steps:



8 Conclusions and future work

In this paper we have sketched a methodology for the formal development of programs supported by the modularisation facilities of Standard ML [MacQ 86], [HMT 87]. Our starting point was the specification language Extended ML [ST 85b], [ST 86], [ST 89] which incorporates these facilities. The present work may be viewed as an adaptation to the Extended ML framework of some of the ideas in [Sch 86] amalgamated with our ideas on implementation of specifications [ST 88b] developed in the context of ASL [SWi 83], [ST 88a]. An important principle which unifies all this work is the central role of behavioural equivalence in program specification and development.

We have borrowed from [Sch 86] the technical concepts of universal correctness (Section 4.2) and simple correctness and stability (Section 6) together with the thesis that it is proper to demand that stability be guaranteed by the programming language. We have generalised his results on composition of universally correct functors by allowing interfaces to match up to behavioural equivalence rather than requiring them to match literally (Section 5). We have also provided some results useful for proving that interfaces match up to behavioural equivalence (Section 3). Although these results are sufficient to handle many cases of interest, we regard them as first attempts in this direction; more work remains to be done here, especially in the context of specifications having a non-trivial structure.

From [ST 88b] we take the concept of constructor implementation and the idea that constructors play a central role in program development. As hinted in the conclusion of [ST 88b], constructors correspond to Standard ML functors. In the Extended ML framework developed here we allow such constructors to be specified before they are actually coded. Implementing an Extended ML functor heading by functor decomposition amounts to sketching the entire constructor implementation process for that functor. Because the constructors involved are specified, the correctness of this decomposition may be verified before any code is written.

In this paper we have considered only a restricted subset of the Standard ML core language, excluding features like polymorphism which are not directly available within the standard algebraic framework. A way to circumvent this limitation is to generalise the work presented here to the framework of an arbitrary institution [GB 84] which formalises the informal concept of an arbitrary logical system. We can see no obvious obstacles to such a generalisation — in fact, most of the work has already been done: the basic methodological ideas in [Sch 86] and [ST 88b] were developed to work in an arbitrary institution, and the institution-independent semantics of (a previous version of) Extended ML was given in [ST 86]. It remains to ensure that everything fits together properly.

In contrast, the technical results on proving behavioural consequence in Section 3 are very much specific to the particular institution used here. It would be interesting to investigate the extension of these results in the framework of institutions enriched with some additional structure, such as so-called abstract algebraic institutions [Tar 85], [Tar 86a], which would seem to support the concepts involved.

Once the generalisation to an arbitrary institution is established, we can instantiate it to the context of an institution which covers all the features of Standard ML. The result would be a framework to support the development of programs in full Standard ML. Constructing such an institution is a separate (and very non-trivial) job. There are many features in Standard ML which have not yet been given an adequate algebraic treatment. Even if they could all be treated separately, it may turn out to be difficult to combine them in a single institution. This is an example of the general problem of how to put institutions together addressed in a preliminary way in [GB 84], [Tar 86b] and [HST 89]. It is intriguing to observe that other programming languages can be accommodated in this framework in a similar way; see [SWa 87] where the modularisation facilities of Standard ML were adapted for Prolog by instantiating an institution-independent version of ML modules.

As explained in Section 6, the soundness of our methodology depends on the stability of the target programming language. This must be checked in detail for the subset of ML we use in this paper and for other potential target languages. Even formulating the stability result requires an algebraic-style semantics for the language, as would be given in the definition of the corresponding institution.

The aims of this work are broadly similar to those of work on *rigorous* program development by the VDM school (see e.g. [Jones 80]). VDM is a method for software specification and development, based on the use of explicitly-defined models of software systems, which has been widely applied in practice. However, it lacks formal mathematical foundations and explicit structuring mechanisms. The RAISE project [BDMP 85] is attempting to fill these gaps. This can be seen as converging with our current work which builds on formal mathematical foundations with a strong emphasis on structure of specifications, and attempts to address problems of practical usability. At a technical level, two advantages of our approach are the use of behavioural equivalence which handles the transition between data specification and representation in a more general way than VDM's *retrieve functions*, and the use of institutions to obtain independence from the underlying logical framework and target programming language.

A notion of modular specification related to the one in Extended ML is developed in a series of papers beginning with [EW 85]. The exact relationship is yet to be investigated. The underlying semantic notions seem to be close although there are numerous technical differences and the main issues of concern differ as well. While [EW 85] and later papers mainly discuss the module concept itself and operations for manipulating modules with compatibility results, in Extended ML these are taken as given since they are inherited directly from Standard ML. Recent work on system development in that framework [EFHLP 87] builds around notions of realization and refinement which seem to be based on different intuitions than the ones we try to model here.

The eventual practical feasibility of formal program development depends on the existence of an integrated support system. There is a need for (at least) the following:

- A parser and typechecker for Extended ML specifications

  This would allow specifications to be checked for silly mistakes, and produce abstract syntax trees in a form suitable for processing by other components of the system. It would also provide the information on sharing required to generate appropriate verification conditions for refinement steps.

- A theorem prover

  Most proofs encountered in proving properties of specifications and programs are routine, albeit sometimes long and intricate. This makes them good candidates for automated proof using methods such as those described in [BM 79], but it is important to have the possibility of proceeding interactively as in LCF [GMW 79] if automated methods fail.

- A refinement step verifier

  Given a refinement step to be verified, a number of conjectures to be proved will be generated and fed to the theorem prover. If these conjectures can be proved (either automatically or interactively) then the correctness of the refinement step follows. It will often be unnecessary to consult the theorem prover since interfaces will match exactly. On the other hand, in case a conjecture generated while verifying a refinement step turns out to be non-trivial to prove, it should be possible to leave it aside at least until it becomes apparent that the line of development is the right one.

- A rapid prototyping capability

  In order to ensure that a specification captures all the required properties of a program, it is important to have some way of exploring its consequences as early as possible in the program development process. We do not agree with the idea that the expressive power of the specification languages should be restricted so as to guarantee that all specifications are "executable". However, we can take advantage of the technology developed in systems like OBJ2 [FGJM 85], REVE [Les 83] and RAP [Hus 85] to allow specifications which happen to be in the appropriate form to be tested. The consequences of specifications not in this form can be explored using the theorem prover.

- Environmental tools

  An advantage of adopting a specification language which is a variant of Standard ML is that we will be able to make use of environmental tools for Standard ML (structure editors, version control, cross-referencing facilities, etc.) as they become available. However, some tools which are particular to the formal program development enterprise will be needed, including for example some mechanism for keeping track of verified refinement steps.

Most of the technology on which such a system depends has already been developed so that constructing it would mostly be a matter of applying and integrating existing techniques rather than inventing new ones.

One thing which is not at all clear is how such a system can be made to accommodate the instantiation of Extended ML and our program development methodology to different institutions. It seems clear that some parts of the system are very much specific to particular logical systems, for example the parser and nearly everything concerned with rapid prototyping. Some other aspects will generalise easily, for example the refinement step verifier, although it is still open how exactly this will work in practice. The problem of generalising specific techniques to arbitrary logical systems has been addressed in a number of other research projects; for example, theorem provers which work in arbitrary logics include EFS [Gri 87] based on the Edinburgh Logical Framework [HHP 87] and Isabelle [Pau 86]. The relation between institutions and LF is under investigation; see [HST 89] for the current status of this work.

Acknowledgements

The work presented here is based on the ideas developed by Oliver Schoett in [Sch 86]. Thanks to Jordi Farrés, Mike Fourman and the anonymous TAPSOFT'89 referees for comments on a draft. This work has been partially supported by grants from the Polish Academy of Sciences and from the U.K. Science and Engineering Research Council. The bulk of the work was done during a visit by the second author to Edinburgh University under the support of an SERC Visiting Fellowship.

9 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[Bau 85] Bauer, F.L. *et al* (the CIP language group). *The Wide Spectrum Language CIP-L.* LNCS 183 (1985).

[BDMP 85] Bjørner, D., Denvir, T., Meiling, E. and Pedersen, J.S. The RAISE project: fundamental issues and requirements. Report RAISE/DDC/EM/1/V6, Dansk Datamatic Center (1985).

[BM 79] Boyer, R.S. and Moore, J.S. *A Computational Logic.* Academic Press (1979).

[BPW 84] Broy, M., Pair, C. and Wirsing, M. A systematic study of models of abstract data types. *Theoretical Computer Science 33*, pp. 139–174 (1984).

[BW 82] Broy, M. and Wirsing, M. Partial abstract types. *Acta Informatica 18* pp. 47–64 (1982).

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of CLEAR, a specification language. *Proc. of Advanced Course on Abstract Software Specification*, Copenhagen, LNCS 86, pp. 292–332 (1980).

[BG 82] Burstall, R.M. and Goguen, J.A. Algebras, theories and freeness: an introduction for computer scientists. *Proc. 1981 Marktoberdorf NATO Summer School.* Reidel (1982).

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

[EFHLP 87] Ehrig, H., Fey, W., Hansen, H., Löwe, M. and Parisi-Presicce, F. Algebraic theory of modular specification development. Technical report, Technische Univ. Berlin (1987).

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. *Theoretical Computer Science 20*, 209–263 (1982).

[EM 85] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer (1985).

[EW 85] Ehrig, H. and Weber, H. Algebraic specification of modules. In: *Formal Models in Programming* (E.J. Neuhold and G. Chronist, eds.). North-Holland, 231–258 (1985).

[Far 89] Farrés-Casals, J. Proving correctness of constructor implementations. Report ECS-LFCS-89-72, University of Edinburgh (1989).

[FGJM 85] Futatsugi, K., Goguen, J.A., Jouannaud, J.-P. and Meseguer, J. Principles of OBJ2. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 52–66 (1985).

[Gan 83] Ganzinger, H. Parameterized specifications: parameter passing and implementation with respect to observability. *Trans. Prog. Lang. Syst. 5*, pp. 318–354 (1983).

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Gdańsk. LNCS 45, pp. 576–587 (1976).

[Gog 84] Goguen, J.A. Parameterized programming. *IEEE Trans. Software Engineering SE-10*, pp. 528–543 (1984).

[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, Computer Science Laboratory, SRI International (1980).

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon, LNCS 164, pp. 221–256 (1984).

[GM 82] Goguen, J.A. and Meseguer, J. Universal realization, persistent interconnection and implementation of abstract modules. *Proc. 9th Intl. Colloq. on Automata, Languages and Programming*, Aarhus. LNCS 140, pp. 265–281 (1982).

[GTW 78] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness and implementation of abstract data types, *Current Trends in Programming Methodology, Vol. IV* (R.T. Yeh, ed.), pp. 80–149, Prentice-Hall (1978).

[GMW 79] Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. *Edinburgh LCF.* LNCS 78 (1979).

[Gri 87] Griffin, T. An environment for formal systems. Report ECS-LFCS-87-34, University of Edinburgh (1987).

[Jones 80] Jones, C. *Software Development: A Rigorous Approach.* Prentice-Hall (1980).

[Har 86] Harper, R. Introduction to Standard ML. Report ECS-LFCS-86-14, University of Edinburgh (1986).

[HHP 87] Harper, R., Honsell, F. and Plotkin, G. A framework for defining logics. *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Cornell (1987).

[HMM 86] Harper, R., MacQueen, D.B. and Milner, R. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[HMT 87] Harper, R., Milner, R. and Tofte, M. The semantics of Standard ML (version 1). Report ECS-LFCS-87-36, Univ. of Edinburgh (1987).

[HMT 88] Harper, R., Milner, R. and Tofte, M. The definition of Standard ML (version 2). Report ECS-LFCS-88-62, Univ. of Edinburgh (1988).

[HST 89] Harper, R., Sannella, D. and Tarlecki, A. Structure and representation in LF. *Proc. 4th IEEE Symp. on Logic in Computer Science*, Asilomar, California, to appear (1989).

[Hoa 72] Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1*, pp. 271–281 (1972)

[Hus 85] Hussmann, H. Rapid prototyping for algebraic specifications: RAP system user's manual. Report MIP-8504, Univ. of Passau (1985).

[Les 83] Lescanne, P. Computer experiments with the REVE term rewriting system generator. *Proc. 10th ACM Symp. on Principles of Programming Languages*, Austin, pp. 99–108 (1983).

[Lip 83] Lipeck, U. Ein algebraischer Kalkül für einer strukturierten Entwurf von Datenabstraktionen. Ph.D. thesis, Abteilung Informatik, Universität Dortmund (1983).

[LB 77] Liskov, B.H. and Berzins, V. An appraisal of program specifications. Computation Structures Group memo 141-1, Laboratory for Computer Science, MIT (1977).

[MacQ 86] MacQueen, D.B. Modules for Standard ML. In [HMM 86] (1986).

[Mil 86] Milner, R. The Standard ML core language (revised). In [HMM 86] (1986).

[Moo 56] Moore, E.F. Gedanken-experiments on sequential machines. In: *Automata Studies* (C.E. Shannon and J. McCarthy, eds.), Princeton Univ. Press, pp. 129–153 (1956).

[NO 88] Nivela, M.P. and Orejas, F. Initial behaviour semantics for algebraic specifications. *Selected Papers from 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland, LNCS 332, pp. 184–207 (1988).

[Ore 83] Orejas, F. Characterizing composability of abstract interpretations. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden, LNCS 158, pp. 335–346 (1983).

[Par 72] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM 15*, pp. 1053–1058 (1972).

[Pau 86] Paulson, L.C. Natural deduction proof as higher-order resolution. *J. of Logic Programming 3*, pp. 237–258 (1986).

[Rei 81] Reichel, H. Behavioural equivalence — a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, Budapest, pp. 27–39 (1981).

[Rei 84] Reichel, H. Behavioural validity of equations in abstract data types. *Contributions to General Algebra 3, Proc. of the Vienna Conference.* Teubner, pp. 301–324 (1984).

[ST 85a] Sannella, D. and Tarlecki, A. Some thoughts on algebraic specification. *Proc. 3rd Workshop on Theory and Applications of Abstract Data Types*, Bremen. Springer Informatik-Fachberichte Vol. 116, pp. 31–38 (1985).

[ST 85b] Sannella, D. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67–77 (1985).

[ST 86] Sannella, D. and Tarlecki, A. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, LNCS 240, pp. 364–389 (1986).

[ST 87] Sannella, D. and Tarlecki, A. On observational equivalence and algebraic specification. *J. Comp. and Sys. Sciences 34*, pp. 150–178 (1987).

[ST 88a] Sannella, D. and Tarlecki, A. Specifications in an arbitrary institution. *Information and Computation 76*, pp. 165–210 (1988).

[ST 88b] Sannella, D.T. and Tarlecki, A. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica 25*, pp. 233–281 (1988). Extended abstract in *Proc. 12th Colloq. on Trees in Algebra and Programming*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Pisa. LNCS 249, pp. 96–110 (1987).

[ST 89] Sannella, D. and Tarlecki, A. The semantics of Extended ML. Research report, Univ. of Edinburgh (in preparation).

[SWa 87] Sannella, D.T. and Wallen, L.A. A calculus for the construction of modular Prolog programs. *Proc. 1987 IEEE Symp. on Logic Programming*, San Francisco, pp. 368–378 (1987).

[SWi 83] Sannella, D. and Wirsing, M. A kernel language for algebraic specification and implementation. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden, LNCS 158, pp. 413–427 (1983).

[Sch 86] Schoett, O. Data abstraction and the correctness of modular programming. Ph.D. thesis, Univ. of Edinburgh (1986).

[Tar 85] Tarlecki, A. On the existence of free models in abstract algebraic institutions. *Theor. Comp. Science 37*, pp. 269–304 (1985).

[Tar 86a] Tarlecki, A. Quasi-varieties in abstract algebraic institutions. *J. Comp. and Sys. Sciences 33*, pp. 333–360 (1986).

[Tar 86b] Tarlecki, A. Bits and pieces of the theory of institutions. *Proc. Workshop on Category Theory and Computer Programming*, LNCS 240, pp. 334–363 (1986).

[Wand 82] Wand, M. Specification, models, and implementations of data abstractions. *Theoretical Computer Science 20* pp. 3–32 (1982).

[Wir 86] Wirsing, M. Structured algebraic specifications: a kernel language. *Theor. Comp. Science 42*, pp. 123–249 (1986).

A Two technical lemmas about behavioural equivalence

The following two rather technical facts allow us to prove satisfaction up to behavioural equivalence in some important specific situations.

Lemma A.1 *Let*

$$\begin{tikzcd}
\Sigma 1 \arrow[r, "\iota_2'"] & \Sigma \\
\Sigma_{shr} \arrow[u, "\iota_1"] \arrow[r, "\iota_2"'] & \Sigma 2 \arrow[u, "\iota_1'"']
\end{tikzcd}$$

be a pushout in the category of algebraic signatures. Let $OBS1 \subseteq sorts(\Sigma 1)$ *and* $OBS2 \subseteq sorts(\Sigma 2)$ *be sets of observable sorts that contain the entire common part* Σ_{shr} *of* $\Sigma 1$ *and* $\Sigma 2$*, i.e. such that* $sorts(\iota_1(\Sigma_{shr})) \subseteq OBS1$ *and* $sorts(\iota_2(\Sigma_{shr})) \subseteq OBS2$*. Consider* $A1, A1' \in Alg(\Sigma 1)$ *and* $A2, A2' \in Alg(\Sigma 2)$ *such that* $A1|_{OBS1} = A1'|_{OBS1}$ *and* $A2|_{OBS1} = A2'|_{OBS2}$*, and moreover* $A1\big|_{\iota_1} = A1'\big|_{\iota_1} = A2\big|_{\iota_2} = A2'\big|_{\iota_2}$*. Let* $A, A' \in Alg(\Sigma)$ *be the amalgamated union of* $A1$ *and* $A2$ *and of* $A1'$ *and* $A2'$ *respectively, i.e.* $A\big|_{\iota_2'} = A1$*,* $A\big|_{\iota_1'} = A2$ *and* $A'\big|_{\iota_2'} = A1'$*,* $A'\big|_{\iota_1'} = A2'$*. Then, if* $A1 \equiv_{OBS1} A1'$ *and* $A2 \equiv_{OBS2} A2'$ *then* $A \equiv_{OBS} A'$*, where* $OBS = \iota_2'(OBS1) \cup \iota_1'(OBS2)$*.*

Proof Let X be an OBS-sorted set of variables, and let $v : X \to |A|_{OBS}$ $(= |A'|_{OBS})$ be a valuation. By the construction of pushouts in the category of algebraic signatures, for any $s \in OBS$ and any $s1 \in sorts(\Sigma 1)$ such that $\iota_2'(s_1) = s$ (resp., $s2 \in sorts(\Sigma 2)$ such that $\iota_1'(s_2) = s$), $s_1 \in OBS1$ (resp. $s_2 \in OBS2$). Consider $t \in T_\Sigma(X)_s$, $s \in OBS$. We have to prove that $t^A(v) = t^{A'}(v)$. We proceed by induction on the structure of t:

Case 1: There is a term $t_1 \in T_{\Sigma 1}(X1)_{s_1}$ such that $\iota_2'(s_1) = s$ and $\iota_2'(t_1) = t$, where $X1$ is an $OBS1$-sorted set of variables such that $X1_r \subseteq X_{\iota_2'(r)}$ for $r \in OBS1$. By an obvious sublemma of the Satisfaction Lemma, $t^A(v) = t_1^{A|_{\iota_2'}}(v1) = t_1^{A1}(v1)$ and $t^{A'}(v) = t_1^{A'|_{\iota_2'}}(v1) = t_1^{A1'}(v1)$ where $v1 : X1 \to |A1|_{OBS1}$ $(= |A1'|_{OBS1})$ is defined by: for $r \in OBS1$, $x \in X1_r$, $v1_r(x) = v_{\iota_2'}(x)$. Now, since $A1 \equiv_{OBS1} A1'$, $t_1^{A1'}(v1) = t_1^{A1}(v1)$, and so indeed $t^A(v) = t^{A'}(v)$.

Case 2: There is a term $t_2 \in T_{\Sigma 2}(X2)_{s_2}$ such that $\iota_1'(s_2) = s$ and $\iota_1'(t_2) = t$, where $X2$ is an $OBS2$-sorted set of variables such that $X2_r \subseteq X_{\iota_1'(r)}$. Proof as above.

Case 3: Otherwise, t must have a subterm, which is not a variable, of a sort in $\iota(\Sigma_{shr})$ which satisfies one of the above conditions. Without loss of generality, assume that t has a non-trivial subterm $t' \in T_\Sigma(X)_{s'}$ such that there is a term $t_1' \in T_{\Sigma 1}(X1)_{s_1}$ where $\iota_2'(s_1) = s'$ and $\iota_2'(t_1') = t'$, and $X1$ is an $OBS1$-sorted set of variables, $X1_r \subseteq X_{\iota_2'(r)}$.

We can rewrite t as $\hat{t}[t'/x]$ where $\hat{t} \in T_\Sigma(X \cup \{x{:}s'\})_s$. Now, as in case 1 above, $t'^A(v) = t'^{A'}(v)$. Let $\hat{v} : X \cup \{x{:}s'\} \to |A|_{OBS}$ $(= |A'|_{OBS})$ be an extension of v given by $\hat{v}(x) = t_1^A(v)$. We have $t^A(v) = (\hat{t}[t'/x])^A(v) = \hat{t}^A(\hat{v})$ and $t^{A'}(v) = (\hat{t}[t'/x])^{A'}(v) = \hat{t}^{A'}(\hat{v})$. But by the induction hypothesis $\hat{t}^A(\hat{v}) = \hat{t}^{A'}(\hat{v})$, and thus $t^A(v) = t^{A'}(v)$.

□

(Another proof may be extracted from the proofs of Proposition 5.2.2 and 5.3.2 (and Theorem 4.4.6) in [Sch 86].)

Lemma A.2 *Consider the following commutative diagram:*

$$\begin{tikzcd}
\Sigma & \\
\Sigma 1 \arrow[u, "\sigma"] & \\
\Sigma_{shr} \arrow[u, "\iota"] \arrow[r, "\rho"'] & \Sigma 2 \arrow[ul, "\tau"']
\end{tikzcd}$$

(i.e. $\rho;\tau = \iota$). Let $SP1, SP2$ be specifications with $Sig[SP1] = \Sigma 1$ and $Sig[SP2] = \Sigma 2$ such that $SP1 \models_{\tau}^{sorts(\rho(\Sigma_{shr}))} SP2$ (that is, **derive from** *$SP1$* **by** *$\tau \models_{\Sigma 2}^{sorts(\rho(\Sigma_{shr}))} SP2$). Then, let $OBS \subseteq sorts(\Sigma_{shr})$ be a set of observable sorts such that all signature morphisms considered are identities on OBS, and let $A \in Alg(\Sigma)$ be an algebra such that $A \models^{OBS}$* **translate** *$SP1$* **by** *σ. Construct a pushout:*

$$\begin{tikzcd}
\Sigma \arrow[r, "\rho'"] & \Sigma' \\
\Sigma_{shr} \arrow[u, "\iota;\sigma"] \arrow[r, "\rho"'] & \Sigma 2 \arrow[u, "\sigma 1"']
\end{tikzcd}$$

Consider the amalgamated union of A and $A\big|_{\tau;\sigma}$, written $A + A\big|_{\tau;\sigma}$, i.e. the unique Σ'-algebra such that $(A + A\big|_{\tau;\sigma})\big|_{\rho'} = A$ and $(A + A\big|_{\tau;\sigma})\big|_{\sigma 1} = A\big|_{\tau;\sigma}$. Then, $A + A\big|_{\tau;\sigma} \models^{OBS}$ **translate** *$SP2$* **by** *$\sigma 1$. Moreover, if $A \equiv_{OBS} B$ and $B\big|_{\sigma} \models SP1$, then there exists a Σ'-algebra B' such that $B'\big|_{\rho'} = B$, $(A + A\big|_{\tau;\sigma}) \equiv_{OBS} B'$ and $B'\big|_{\sigma 1} \models SP2$.*

Proof Consider an arbitrary Σ-algebra B such that $B \equiv_{OBS} A$ and $B \models$ **translate** $SP1$ **by** σ, i.e. $B\big|_{\sigma} \models SP1$. Such an algebra B exists since $A \models^{OBS}$ **translate** $SP1$ **by** σ. Since $SP1 \models_{\Sigma 2}^{sorts(\rho(\Sigma_{shr}))} SP2$, $B\big|_{\tau;\sigma} \models_{\Sigma 2}^{sorts(\rho(\Sigma_{shr}))} SP2$, i.e. there exists a model $C \in Mod[SP2]$ such that $B\big|_{\tau;\sigma} \equiv_{sorts(\rho(\Sigma_{shr}))} C$. In particular, $B\big|_{\iota;\sigma} = (B\big|_{\tau;\sigma})\big|_{\rho} = C\big|_{\rho}$. Consider the amalgamated union $B' = B + C$, i.e. the unique Σ'-algebra such that $B'\big|_{\rho'} = B$ and $B'\big|_{\sigma 1} = C$. Clearly, $B' \models$ **translate** $SP2$ **by** $\sigma 1$. To complete the proof that $A + A\big|_{\tau;\sigma} \models^{OBS}$ **translate** $SP2$ **by** $\sigma 1$, we will show that $B' \equiv_{OBS} A + A\big|_{\tau;\sigma}$.

By the construction, there exists a unique algebraic signature morphism $\tau' : \Sigma' \to \Sigma$ such that the following diagram commutes:

$$\begin{tikzcd}
 & & \Sigma \\
\Sigma \arrow[urr, "id"] \arrow[r, "\rho'"'] & \Sigma' \arrow[ur, "\tau'"'] & \\
\Sigma_{shr} \arrow[u, "\iota;\sigma"] \arrow[r, "\rho"'] & \Sigma 2 \arrow[u, "\sigma 1"] \arrow[uur, "\tau;\sigma"'] &
\end{tikzcd}$$

That is, $\rho';\tau' = id$ and $\sigma 1;\tau' = \tau;\sigma$.

Warning The following diagram does *not* commute in general:

$$\begin{tikzcd}
\Sigma \arrow[r, shift right, "\rho'"'] & \Sigma' \arrow[l, shift right, "\tau'"'] \\
\Sigma 1 \arrow[u, "\sigma"] & \\
\Sigma_{shr} \arrow[u, "\iota"] \arrow[r, "\rho"'] & \Sigma 2 \arrow[ul, "\tau"'] \arrow[uu, "\sigma 1"']
\end{tikzcd}$$

In general, $\tau';\rho' \neq id$. Intuitively, to construct Σ' we took the signature Σ and then removed all identification of symbols coming from $\Sigma 2$ which are not inherited from Σ_{shr}. Consequently, $A + A\big|_{\tau;\sigma}$ is just the "same" algebra as A, but with some types and values duplicated rather than shared. $\square$(Warning)

Sublemma *For any Σ-algebra D, $D\big|_{\tau'} = D + D\big|_{\tau;\sigma}$.*

Proof Just notice that $(D\big|_{\tau'})\big|_{\rho'} = D\big|_{\rho';\tau'} = D\big|_{id} = D$ and $(D\big|_{\tau'})\big|_{\sigma 1} = D\big|_{\sigma 1;\tau'} = D\big|_{\tau;\sigma}$. $\square$

Now, since $A \equiv_{OBS} B$, we also have $A\big|_{\tau'} \equiv_{OBS} B\big|_{\tau'}$, i.e. $(A+A\big|_{\tau;\sigma}) \equiv_{OBS} (B+B\big|_{\tau;\sigma})$. However, by Lemma A.1, $(B+B\big|_{\tau;\sigma}) \equiv_{sorts(\sigma 1(\rho(\Sigma_{shr})))} B'$ which implies $B+B\big|_{\tau;\sigma} \equiv_{OBS} B'$ and so $A+A\big|_{\tau;\sigma} \equiv_{OBS} B'$. $\square$(Lemma A.2)

B Unitary decomposition theorem

Theorem 5.1 *Consider an Extended ML functor F:*

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl} = F1(X)$$

where $F1$ is a functor with heading:

$$\texttt{functor}\ F1(X : SIG1_{in}) : SIG1_{out}\ \texttt{sharing}\ \textit{sharing-decl}1$$

Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, determining a commutative diagram as in Section 5.1. Suppose the following conditions are satisfied:

1. $SIG_{in} \models_{\Sigma 1_{in}}^{sorts(\iota 1_{in}(\rho(\Sigma_{shr})))} SIG1_{in}$
2. $SIG1_{out} \models_{\Sigma_{out}}^{sorts(\iota_{out}(\Sigma_{shr}))} SIG_{out}$

Then, if $F1$ is universally correct then F is universally correct.

Proof Let $F1_{bsem} \in \mathcal{F}1_{bsem}$. The corresponding basic semantics of F, $F_{bsem} : Alg(\Sigma_{in}) \rightsquigarrow Alg(\Sigma_{out})$ is defined as follows: for any $A \in Alg(\Sigma_{in})$, $F_{bsem}(A) = F1_{bsem}(A\big|_{\tau_{in}})\big|_{\tau_{out}}$[13]. (Since the diagram

[13]Here, and in similar situations, such a definitional equation implicitly says that the left-hand side is defined if and only if the right-hand side is defined.

of Section 5.1 commutes, the fact that $A|_{\Sigma_{shr}} = F_{bsem}(A)|_{\Sigma_{shr}}$ follows easily from $(A|_{\tau_{in}})|_{\Sigma 1_{shr}} = F1_{bsem}(A|_{\tau_{in}})|_{\Sigma 1_{shr}}$.)

Consider any algebraic signature Σ_{arg}, fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$, and Σ_{arg}-algebra A. Suppose that $A \models^{sorts(\mathtt{Perv})}$ **translate** SIG_{in} **by** σ. We have to prove that

$$A|_{\sigma} \in Dom(F) \quad \text{and} \quad F_{gres}(A[\sigma]) \models^{sorts(\mathtt{Perv})} \textbf{translate } SIG_{out} \textbf{ by } F[\sigma]$$

where the signature of $F_{gres}(A[\sigma])$ and the morphism $F[\sigma]$ are defined by the pushout:

$$\begin{tikzcd}
\Sigma_{arg} \arrow[r, "\iota'_{out}"] & F(\Sigma_{arg}[\sigma]) \\
\Sigma_{in} \arrow[u, "\sigma"] & \\
\Sigma_{shr} \arrow[u, "\iota_{in}"] \arrow[r, "\iota_{out}"'] & \Sigma_{out} \arrow[uu, "F[\sigma]"']
\end{tikzcd}$$

and $F_{gres}(A[\sigma]) = A + F_{bsem}(A|_{\sigma})$ is the amalgamated union as in Section 4.1.

Let the following diagram be a pushout in the category of algebraic signatures:

$$\begin{tikzcd}
\Sigma_{arg} \arrow[r, "\rho'"] & \Sigma 1_{arg} \\
\Sigma_{in} \arrow[u, "\sigma"] & \\
\Sigma_{shr} \arrow[u, "\iota_{in}"] \arrow[r, "\rho"'] & \Sigma 1_{in} \arrow[uu, "\sigma 1"']
\end{tikzcd}$$

Consider the amalgamated union $A1 =_{\text{def}} (A + A|_{\tau_{in};\sigma}) \in Alg(\Sigma 1_{arg})$. By Lemma A.2, $A1 \models^{sorts(\mathtt{Perv})}$ **translate** $SIG1_{in}$ **by** $\sigma 1$. Hence, we can apply $F1$ to $A + A|_{\tau_{in};\sigma}$ using the fitting morphism $\sigma 1$. In particular, $A1|_{\sigma 1} = (A|_{\sigma})|_{\tau_{in}} \in Dom(F1)$ by universal correctness of $F1$ and hence $A|_{\sigma} \in Dom(F)$. By definition, we obtain the following pushout diagram:

$$\begin{tikzcd}
\Sigma 1_{arg} \arrow[r, "\iota 1'_{out}"] & F1(\Sigma 1_{arg}[\sigma 1]) \\
\Sigma 1_{in} \arrow[u, "\sigma 1"] & \\
\Sigma 1_{shr} \arrow[u, "\iota 1_{in}"] \arrow[r, "\iota 1_{out}"'] & \Sigma 1_{out} \arrow[uu, "F1[\sigma 1]"']
\end{tikzcd}$$

and the global result of $F1$, which is the amalgamated union $F1_{gres}(A1[\sigma 1]) = A1 + F1_{bsem}(A|_{\tau_{in};\sigma})$ (recall that $A1|_{\sigma 1} = (A + A|_{\tau_{in};\sigma})|_{\sigma 1} = A|_{\tau_{in};\sigma}$). By the universal correctness of $F1$,

$$F1_{gres}(A1[\sigma 1]) \models^{sorts(\mathtt{Perv})} \textbf{translate } SIG1_{out} \textbf{ by } F1[\sigma 1].$$

We still have to coerce the result to the output signature. Let the following diagram be a pushout:

$$
\begin{tikzcd}
F1(\Sigma 1_{arg}[\sigma 1]) \arrow[r, "\rho''"] & \Sigma 2 \\
\Sigma 1_{out} \arrow[u, "F1[\sigma 1]"] & \\
\Sigma 1_{shr} \arrow[u, "\iota 1_{out}"] & \\
\Sigma_{shr} \arrow[u, "\rho"] \arrow[r, "\iota_{out}"'] & \Sigma_{out} \arrow[uuu, "\sigma 2"']
\end{tikzcd}
$$

Consider the amalgamated union $A2 =_{\rm def} (F1_{gres}(A1[\sigma 1]) + F1_{gres}(A1[\sigma 1])\big|_{\tau_{out};F1[\sigma 1]}) \in Alg(\Sigma 2)$. By Lemma A.2, $A2 \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $\sigma 2$.

It may be helpful at this point to study the commutative diagram in the category of algebraic signatures which we have constructed so far:

$$
\begin{tikzcd}
& \Sigma 1_{arg} \arrow[r, "\iota 1'_{out}"] & F1(\Sigma 1_{arg}[\sigma 1]) \arrow[rr, "\rho''"] & & \Sigma 2 \\
\Sigma_{arg} \arrow[ur, "\rho'"] \arrow[rrr, "\iota'_{out}"] & & & F(\Sigma_{arg}[\sigma]) & \\
\Sigma_{in} \arrow[u, "\sigma"] & & & & \\
& \Sigma 1_{in} \arrow[uuu, "\sigma 1"] \arrow[ul, "\tau_{in}"'] & & & \\
& \Sigma 1_{shr} \arrow[u, "\iota 1_{in}"] \arrow[r, "\iota 1_{out}"'] & \Sigma 1_{out} \arrow[uuuu, "F1[\sigma 1]"] & & \\
\Sigma_{shr} \arrow[uuu, "\iota_{in}"] \arrow[ur, "\rho"'] \arrow[rrr, "\iota_{out}"'] & & & \Sigma_{out} \arrow[ul, "\tau_{out}"'] \arrow[uuuu, "F[\sigma]"] \arrow[uuuuur, bend right, "\sigma 2"'] &
\end{tikzcd}
$$

By the construction of $F(\Sigma_{arg}[\sigma])$ as a pushout object, there exists a (unique) morphism $\tau_{fin} : F(\Sigma_{arg}[\sigma]) \to \Sigma 2$ such that $F[\sigma];\tau_{fin} = \sigma 2$ and $\iota'_{out};\tau_{fin} = \rho';\iota 1'_{out};\rho''$. We claim that $A2\big|_{\tau_{fin}} = F_{gres}(A[\sigma])$. To verify this, just consider:

$$
\begin{aligned}
(A2\big|_{\tau_{fin}})\big|_{\iota'_{out}} &= A2\big|_{\iota'_{out};\tau_{fin}} \\
&= (F1_{gres}(A1[\sigma 1]) + F1_{gres}(A1[\sigma 1])\big|_{\tau_{out};F1[\sigma 1]})\big|_{\rho';\iota 1'_{out};\rho''} \\
&= F1_{gres}(A1[\sigma 1])\big|_{\rho';\iota 1'} \\
&= A1\big|_{\rho'} = A
\end{aligned}
$$

and

$$
\begin{aligned}
(A2|_{\tau_{fin}})|_{F[\sigma]} &= A2|_{F[\sigma];\tau_{fin}} \\
&= A2|_{\sigma 2} \\
&= F1_{gres}(A1[\sigma 1])|_{\tau_{out};F1[\sigma 1]} \\
&= (F1_{gres}(A1[\sigma 1])|_{F1[\sigma 1]})|_{\tau_{out}} \\
&= F1_{bsem}(A1|_{\sigma 1})|_{\tau_{out}} \\
&= F1_{bsem}((A|_{\sigma})|_{\tau_{in}})|_{\tau_{out}} \\
&= F_{bsem}(A|_{\sigma})
\end{aligned}
$$

which proves the claim by the construction of $F_{gres}(A[\sigma])$ as an amalgamated union.

Finally, since $A2 \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $\sigma 2$, there exists a $\Sigma 2$-algebra $B2$ such that $B2 \equiv_{sorts(\texttt{Perv})} A2$ and $B2|_{\sigma 2} \models SIG_{out}$. So $F_{gres}(A[\sigma]) = A2|_{\tau_{fin}} \equiv_{sorts(\texttt{Perv})} B2|_{\tau_{fin}}$ and $(B2|_{\tau_{fin}})|_{F[\sigma]} = B2|_{\sigma 2} \models SIG_{out}$. Thus, it is indeed the case that $F_{gres}(A[\sigma]) \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $F[\sigma]$.

To verify the additional requirement imposed by universal correctness, consider a Σ_{arg}-algebra B such that $B \equiv_{sorts(\texttt{Perv})} A$ and $B|_{\sigma} \models SIG_{in}$. We have to construct $\widehat{B} \in Alg(F(\Sigma_{arg}[\sigma]))$ such that $\widehat{B}|_{\iota'_{out}} = B$, $\widehat{B} \equiv_{sorts(\texttt{Perv})} F_{gres}(A[\sigma])$ and $\widehat{B}|_{F[\sigma]} \models SIG_{out}$. The construction parallels the construction of $A2|_{\tau_{fin}}$. Namely:

- By Lemma A.2, there exists a $\Sigma 1_{arg}$-algebra $B1$ such that $B1|_{\rho'} = B$, $B1 \equiv_{sorts(\texttt{Perv})} A1$ and $B1|_{\sigma 1} \models SIG1_{in}$.
- Since $F1$ is universally correct, there exists an $F1(\Sigma 1_{arg}[\sigma 1])$-algebra $\widehat{B1}$ such that $\widehat{B1}|_{\iota 1'_{out}} = B1$, $\widehat{B1} \equiv_{sorts(\texttt{Perv})} F1_{gres}(A1[\sigma 1])$ and $\widehat{B1}|_{F1[\sigma 1]} \models SIG1_{out}$.
- By Lemma A.2, there exists a $\Sigma 2$-algebra $B2$ such that $B2|_{\rho''} = \widehat{B1}$, $B2 \equiv_{sorts(\texttt{Perv})} A2$ and $B2|_{\sigma 2} \models SIG_{out}$.

Finally, let $\widehat{B} = B2|_{\tau_{fin}}$. Then $\widehat{B}|_{F[\sigma]} = B2|_{F[\sigma];\tau_{fin}} = B2|_{\sigma 2} \models SIG_{out}$ and $\widehat{B}|_{\iota'_{out}} = B2|_{\iota'_{out};\tau_{fin}} = B2|_{\rho';\iota 1'_{out};\rho''} = B$. □

C Sequential decomposition theorem

Theorem 5.2 *Consider an Extended ML functors F:*

$$\texttt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \texttt{sharing}\ \textit{sharing-decl} = G2(G1(X))$$

where $G1$ and $G2$ are functors with headings:

$$\texttt{functor}\ G1(Y1 : SIG1_{in}) : SIG1_{out}\ \texttt{sharing}\ \textit{sharing-decl}1$$
$$\texttt{functor}\ G2(Y2 : SIG2_{in}) : SIG2_{out}\ \texttt{sharing}\ \textit{sharing-decl}2$$

Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, determining a commutative diagram as in Section 5.2. Suppose that the following conditions are satisfied:

1. $SIG_{in} \models_{\Sigma 1_{in}}^{sorts(\iota 1_{in}(\rho 1(\Sigma_{shr})))} SIG1_{in}$

2. $SIG1_{out} \models^{sorts(\iota 2_{in}(\rho 2(\Sigma_{shr})))}_{\Sigma 2_{in}} SIG2_{in}$

3. $SIG2_{out} \models^{sorts(\iota_{out}(\Sigma_{shr}))}_{\Sigma_{out}} SIG_{out}$

Then, if $G1$ and $G2$ are universally correct then so is F.

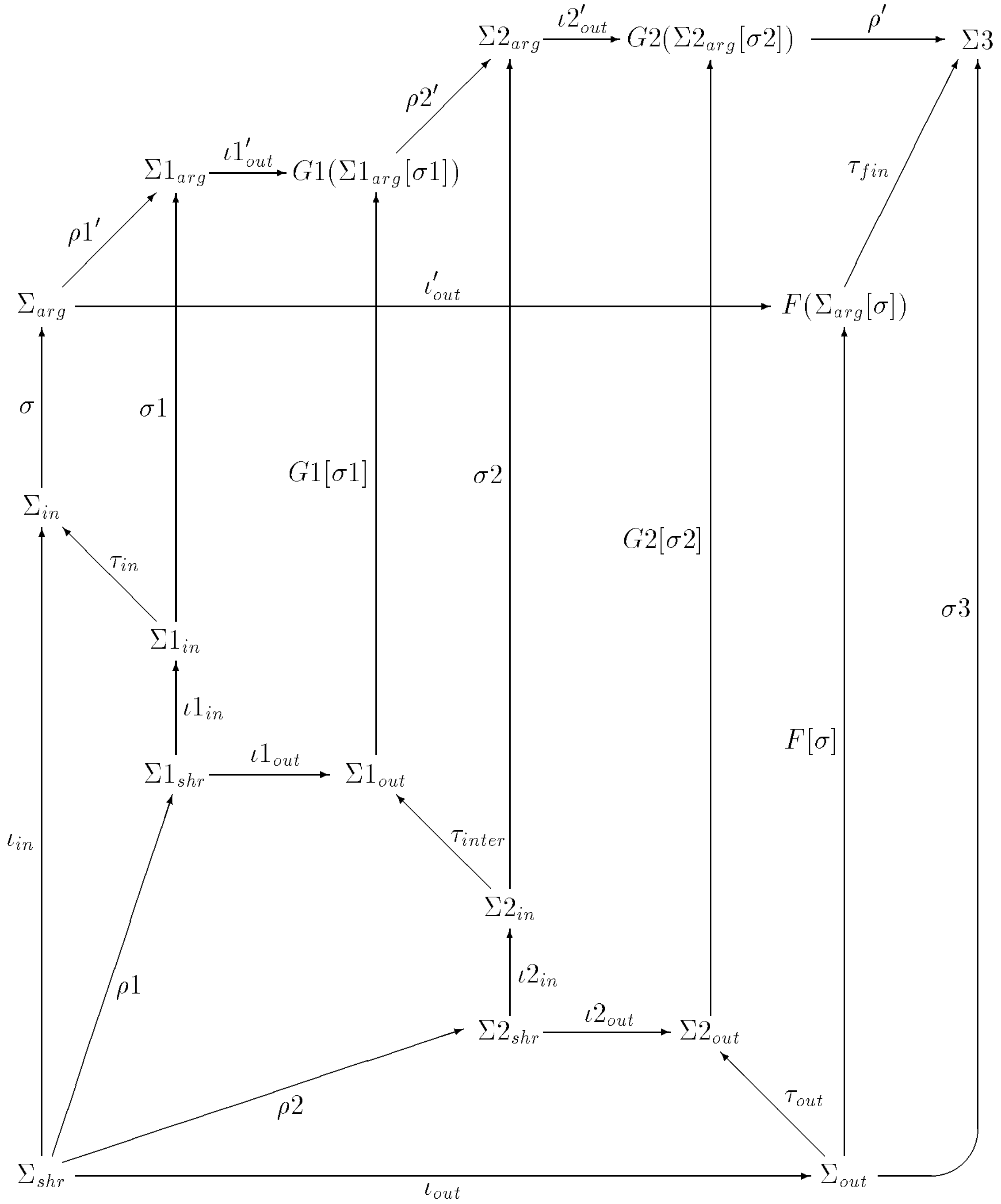
Proof The basic semantics for F is $\mathcal{F}_{bsem} : Alg(\Sigma_{in}) \to \mathcal{P}ow(Alg(\Sigma_{out}))$ defined as follows: for any $A \in Alg(\Sigma_{in})$,

$$\mathcal{F}_{bsem}(A) = \{A2|_{\tau_{out}} \mid A2 \in \mathcal{G}2_{bsem}(A1|_{\tau_{inter}}) \text{ for some } A1 \in \mathcal{G}1_{bsem}(A|_{\tau_{in}})\}$$

or, with a slight abuse of notation,

$$\mathcal{F}_{bsem}(A) = \{G2_{bsem}(G1_{bsem}(A|_{\tau_{in}})|_{\tau_{inter}})|_{\tau_{out}} \mid G1_{bsem} \in \mathcal{G}1_{bsem} \text{ and } G2_{bsem} \in \mathcal{G}2_{bsem}\}.$$

Consider any algebraic (argument) signature Σ_{arg} and fitting morphism $\sigma : \Sigma_{in} \to \Sigma_{arg}$. Proceeding exactly as in the proof of Theorem 5.1, except this time repeating the basic construction twice, we obtain the following commutative diagram in the category of algebraic signatures:



In the above, the following diagrams are defined to be pushouts in the category of algebraic signatures (each pushout is presented by naming the nodes on its two paths; the last morphism in each path results from the pushout construction):

- $\Sigma_{shr} \hookrightarrow \Sigma_{in} \to \Sigma_{arg} \to \Sigma 1_{arg}$ and
  $\Sigma_{shr} \to \Sigma 1_{shr} \hookrightarrow \Sigma 1_{in} \to \Sigma 1_{arg}$,

- $\Sigma 1_{shr} \hookrightarrow \Sigma 1_{in} \to \Sigma 1_{arg} \to G1(\Sigma 1_{arg}[\sigma 1])$ and
  $\Sigma 1_{shr} \hookrightarrow \Sigma 1_{out} \to G1(\Sigma_{arg}[\sigma 1])$,

- $\Sigma_{shr} \to \Sigma 1_{shr} \hookrightarrow \Sigma 1_{out} \to G1(\Sigma 1_{arg}[\sigma 1]) \to \Sigma 2_{arg}$ and
  $\Sigma_{shr} \to \Sigma 2_{shr} \hookrightarrow \Sigma 2_{in} \to \Sigma 2_{arg}$,

- $\Sigma 2_{shr} \hookrightarrow \Sigma 2_{in} \to \Sigma 2_{arg} \to G2(\Sigma 2_{arg}[\sigma 2])$ and
  $\Sigma 2_{shr} \hookrightarrow \Sigma 2_{out} \to G2(\Sigma 2_{arg}[\sigma 2])$,

- $\Sigma_{shr} \to \Sigma 2_{shr} \hookrightarrow \Sigma 2_{out} \to G2(\Sigma 2_{arg}[\sigma 2]) \to \Sigma 3$ and
  $\Sigma_{shr} \hookrightarrow \Sigma_{out} \to \Sigma 3$

- $\Sigma_{shr} \hookrightarrow \Sigma_{in} \to \Sigma_{arg} \to F(\Sigma_{arg}[\sigma])$ and
  $\Sigma_{shr} \hookrightarrow \Sigma_{out} \to F(\Sigma_{arg}[\sigma])$,

and then $\tau_{fin} : F(\Sigma_{arg}[\sigma]) \to \Sigma 3$ is the unique algebraic signature morphism such that

$$\iota'_{out};\tau_{fin} = \rho 1';\iota 1'_{out};\rho 2';\iota 2'_{out};\rho' \qquad \text{and} \qquad F[\sigma];\tau_{fin} = \sigma 3.$$

Now, consider any Σ_{arg}-algebra A such that $A \models^{sorts(\texttt{Perv})}$ **translate** SIG_{in} **by** σ. For any choice of $G1_{bsem} \in \mathcal{G}1_{bsem}$ and $G2_{bsem} \in \mathcal{G}2_{bsem}$ we proceed as follows:

$A1 =_{def} (A + A|_{\tau_{in};\sigma}) \in Alg(\Sigma 1_{arg})$,

then $A1 \models^{sorts(\texttt{Perv})}$ **translate** $SIG1_{in}$ **by** $\sigma 1$ by Lemma A.2.

$G1_{gres}(A1[\sigma 1]) =_{def} (A1 + G1_{bsem}(A|_{\tau_{in};\sigma})) \in Alg(G1(\Sigma 1_{arg}[\sigma 1]))$,

then $G1_{gres}(A1[\sigma 1]) \models^{sorts(\texttt{Perv})}$ **translate** $SIG1_{out}$ **by** $G1[\sigma 1]$ by universal correctness of $G1$.

$A2 =_{def} (G1_{gres}(A1[\sigma 1]) + G1_{bsem}(A|_{\tau_{in};\sigma})) \in Alg(\Sigma 2_{arg})$,

then $A2 \models^{sorts(\texttt{Perv})}$ **translate** $SIG2_{in}$ **by** $\sigma 2$ by Lemma A.2.

$G2_{gres}(A2[\sigma 2]) =_{def} (A2 + G2_{bsem}(G1_{bsem}(A|_{\tau_{in};\sigma}))|_{\tau_{inter}}) \in Alg(G2(\Sigma 2_{arg}[\sigma 2]))$,

then $G2_{gres}(A2[\sigma 2]) \models^{sorts(\texttt{Perv})}$ **translate** $SIG2_{out}$ **by** $G2[\sigma 2]$ by universal correctness of $G2$.

$A3 =_{def} (G2_{gres}(A2[\sigma 2]) + G2_{bsem}(G1_{bsem}(A|_{\tau_{in};\sigma}))|_{\tau_{out}}) \in Alg(\Sigma 3_{arg})$,

then $A3 \models^{sorts(\texttt{Perv})}$ **translate** SIG_{out} **by** $\sigma 3$ by Lemma A.2.

Moreover, we can verify that $(A3|_{\tau_{fin}})|_{\iota'_{out}} = A$ and $(A3|_{\tau_{fin}})|_{F[\sigma]} = G2_{bsem}(G1_{bsem}((A|_{\sigma})|_{\tau_{in}})|_{\tau_{inter}})|_{\tau_{out}}$, which shows that $A|_{\sigma} \in Dom(F)$. We conclude:

$$F_{gres}(A[\sigma]) \models^{sorts(\texttt{Perv})} \textbf{translate } SIG_{out} \textbf{ by } F[\sigma].$$

Finally, let $B \in Alg(\Sigma_{arg})$ such that $B \equiv_{sorts(\texttt{Perv})} A$, $B|_{\sigma} \models SIG_{in}$. In parallel with the above construction we can show the existence of the following algebras:

- $B1 \in Alg(\Sigma 1_{arg})$ such that $B1|_{\rho 1'} = B$, $B1 \equiv_{sorts(\texttt{Perv})} A1$ and $B1|_{\sigma 1} \models SIG1_{in}$ (by Lemma A.2).

- $\widehat{B1} \in Alg(G1(\Sigma 1_{arg}[\sigma 1]))$ such that $\widehat{B1}|_{\iota 1'_{out}} = B1$, $\widehat{B1} \equiv_{sorts(\texttt{Perv})} G1_{gres}(A1[\sigma 1])$ and $\widehat{B1}|_{G1[\sigma 1]} \models SIG1_{out}$ (by universal correctness of $G1$).

- $B2 \in Alg(\Sigma 2_{arg})$ such that $B2|_{\rho 2'} = \widehat{B1}$, $B2 \equiv_{sorts(\texttt{Perv})} A2$ and $B2|_{\sigma 2} \models SIG2_{in}$ (by Lemma A.2).

- $\widehat{B2} \in Alg(G2(\Sigma 2_{arg}[\sigma 2]))$ such that $\widehat{B2}|_{\iota 2'_{out}} = B2$, $\widehat{B2} \equiv_{sorts(\texttt{Perv})} G2_{gres}(A2[\sigma 2])$ and $\widehat{B2}|_{G2[\sigma 2]} \models SIG2_{out}$ (by universal correctness of $G2$).

- $B3 \in Alg(\Sigma 3)$ such that $B3|_{\rho'} = \widehat{B2}$, $B3 \equiv_{sorts(\mathtt{Perv})} A3$ and $B3|_{\sigma 3} \models SIG_{out}$ (by Lemma A.2).

Let $\widehat{B} = B3|_{\tau_{fin}}$. Similarly as in the proof of Theorem 5.1, it is easy to verify that $\widehat{B}|_{\iota'_{out}} = B$ and $\widehat{B}|_{F[\sigma]} \models SIG_{out}$. $\square$

D Parallel decomposition results

Proposition 5.3 *Consider Extended ML functors $G1$ and $G2$ with headings*

$$\texttt{functor } G1(Y1 : SIG1_{in}) : SIG1_{out} \texttt{ sharing } \textit{sharing-decl}1$$
$$\texttt{functor } G2(Y2 : SIG2_{in}) : SIG2_{out} \texttt{ sharing } \textit{sharing-decl}2$$

Let $G12$ be the functor formed by collapsing $G1$ and $G2$ as defined in Section 5.3. Then $G12$ is universally correct provided that $G1$ and $G2$ are.

Proof The semantics of $G12$, as defined in Section 5.3, may be restated as follows: for any $G1_{bsem} \in \mathcal{G}1_{bsem}$ and $G2_{bsem} \in \mathcal{G}2_{bsem}$, there is a corresponding basic semantic function $G12_{bsem} : Alg(\Sigma 12_{in}) \rightsquigarrow Alg(\Sigma 12_{out})$ such that for any $A \in Dom(G12)$, $G12_{bsem}(A)$ is the unique $\Sigma 12_{out}$-algebra defined by

$$G12_{bsem}(A)|_{c1_{out}} = G1_{bsem}(A|_{c1_{in}}) \qquad \text{and} \qquad G12_{bsem}(A)|_{c2_{out}} = G2_{bsem}(A|_{c2_{in}}).$$

This is well-defined, since

$$\begin{aligned}
G1_{bsem}(A|_{c1_{in}})|_{\rho 1_{inter};\iota 1_{out}} &= (G1_{bsem}(A|_{c1_{in}})|_{\iota 1_{out}})|_{\rho 1_{inter}} \\
&= ((A|_{c1_{in}})|_{\iota 1_{in}})|_{\rho 1_{inter}} \\
&= A|_{\rho 1_{inter};\iota 1_{in};c1_{in}} \\
&= A|_{\rho 2_{inter};\iota 2_{in};c2_{in}} \\
&= ((A|_{c2_{in}})|_{\iota 2_{in}})|_{\rho 2_{inter}} \\
&= (G2_{bsem}(A|_{c2_{in}})|_{\iota 2_{out}})|_{\rho 2_{inter}} \\
&= G2_{bsem}(A|_{c2_{in}})|_{\rho 2_{inter};\iota 2_{out}}.
\end{aligned}$$

To verify that for any $A \in Alg(\Sigma 12_{in})$, $A|_{\Sigma 12_{shr}} = G12_{bsem}(A)|_{\Sigma 12_{shr}}$, it is enough to notice that:

$$\begin{aligned}
(G12_{bsem}(A)|_{\iota 12_{out}})|_{c1_{shr}} &= G12_{bsem}(A)|_{c1_{shr};\iota 12_{out}} \\
&= G12_{bsem}(A)|_{\iota 1_{out};c1_{out}} \\
&= (G12_{bsem}(A)|_{c1_{out}})|_{\iota 1_{out}} \\
&= G1_{bsem}(A|_{c1_{in}})|_{\iota 1_{out}} \\
&= (A|_{c1_{in}})|_{\iota 1_{in}} = A|_{\iota 1_{in};c1_{in}} = A|_{c1_{shr};\iota 12_{in}} \\
&= (A|_{\iota 12_{in}})|_{c1_{shr}}
\end{aligned}$$

and similarly

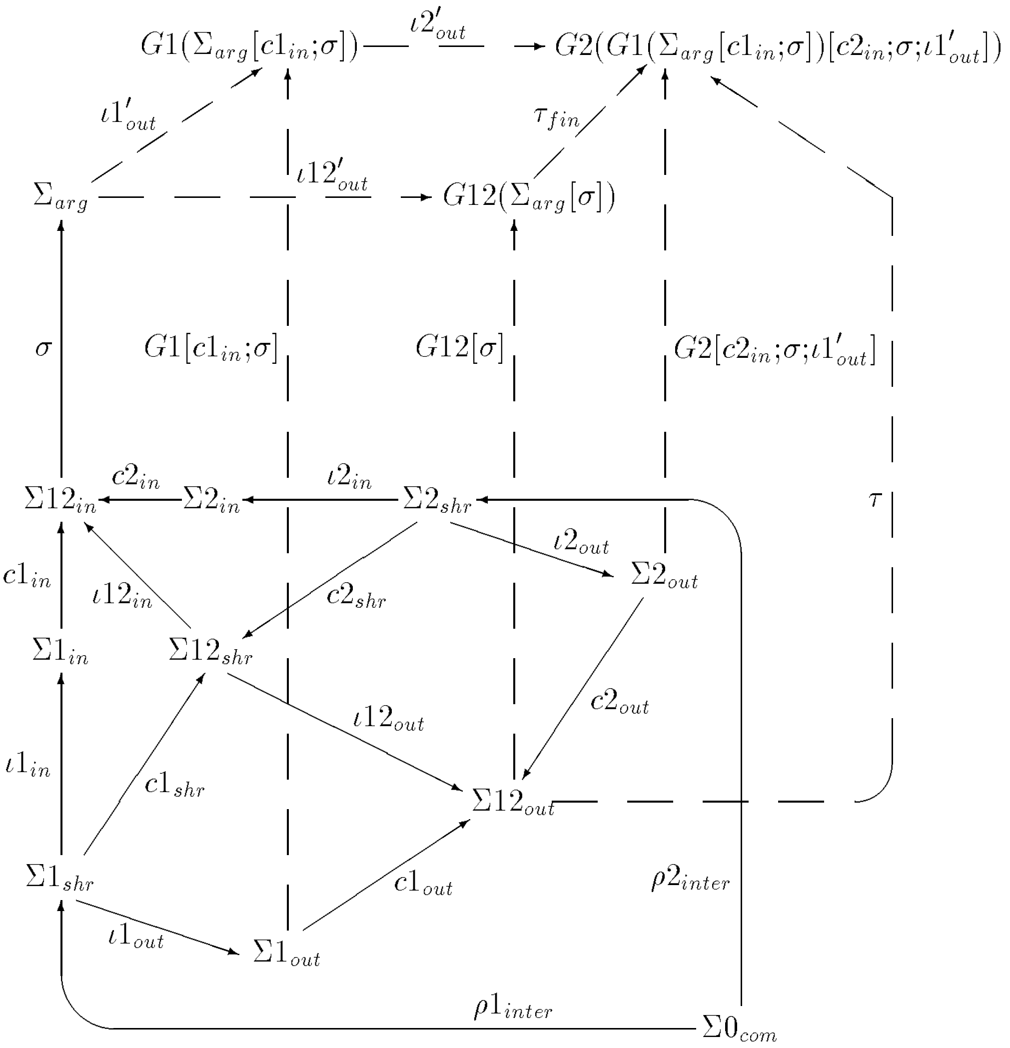
$$(G12_{bsem}(A)|_{\iota 12_{out}})|_{c2_{shr}} = (A|_{\iota 12_{in}})|_{c2_{shr}}.$$

Consider an arbitrary $G1_{bsem} \in \mathcal{G}1_{bsem}$ and $G2_{bsem} \in \mathcal{G}2_{bsem}$ and let $G12_{bsem} \in \mathcal{G}12_{bsem}$ be the corresponding basic semantic function for $G12$.

Consider any algebraic signature Σ_{arg}, fitting morphism $\sigma : \Sigma 12_{in} \to \Sigma_{arg}$ and Σ_{arg}-algebra A such that $A \models^{sorts(\mathtt{Perv})}_{\Sigma_{arg}}$ **translate** $SIG12_{in}$ **by** σ. Let B be any Σ_{arg}-algebra such that $B \equiv_{sorts(\mathtt{Perv})} A$ and

$B|_\sigma \models SIG12_{in}$. We have to show that $A|_\sigma \in Dom(G12)$ and to construct a $G12(\Sigma_{arg}[\sigma])$-algebra $\widehat{B}$ such that $\widehat{B}|_{\iota 12'_{out}} = B$, $\widehat{B} \equiv_{sorts(\mathtt{Perv})} G12_{gres}(A[\sigma])$ and $\widehat{B}|_{G12[\sigma]} \models SIG12_{out}$ (this also implies that $G12_{gres}(A[\sigma]) \models^{sorts(\mathtt{Perv})}$ **translate** $SIG12_{out}$ **by** $G12[\sigma]$).

In the following diagram, dashed arrows denote morphisms constructed in the course of the proof below.



By the definition of $SIG12_{in}$, $B|_{c1_{in};\sigma} \models SIG1_{in}$. Hence, we can safely apply $G1$ to A via the fitting morphism $c1_{in};\sigma : \Sigma 1_{in} \to \Sigma_{arg}$, and the universal correctness of $G1$ ensures that $A|_{c1_{in};\sigma} \in Dom(G1)$ and that there exists a $G1(\Sigma_{arg}[\sigma])$-algebra $B1$ such that $B1|_{\iota 1'_{out}} = B$, $B1 \equiv_{sorts(\mathtt{Perv})} G1_{gres}(A[c1_{in};\sigma])$ and $B1|_{G1[c1_{in};\sigma]} \models SIG1_{out}$.

It follows that $B1|_{c2_{in};\sigma;\iota 1'_{out}} = B|_{c2_{in};\sigma} \models SIG2_{in}$, and so we can safely apply $G2$ to $G1_{gres}(A[c1_{in};\sigma])$ via the fitting morphism $c2_{in};\sigma;\iota 1'_{out} : \Sigma 2_{in} \to G1(\Sigma_{arg}[c1_{in};\sigma])$. Universal correctness of $G2$ ensures that $G1_{gres}(A[c1_{in};\sigma])|_{c2_{in};\sigma;\iota 1'_{out}} \in Dom(G2)$ and that there is a $G2(G1(\Sigma_{arg}[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}])$-algebra $B2$ such that

- $B2|_{\iota 2'_{out}} = B1$;
- $B2 \equiv_{sorts(\mathtt{Perv})} G2_{gres}(G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}])$; and
- $B2|_{G2[c2_{in};\sigma;\iota 1'_{out}]} \models SIG2_{out}$.

Now, since we have

$$\begin{array}{rcl}\rho 1_{inter};\iota 1_{out};G1[c1_{in};\sigma];\iota 2'_{out} & = & \rho 1_{inter};\iota 1_{in};c1_{in};\sigma \iota 1'_{in}\iota 2'_{out} \\ & = & \rho 2_{inter};\iota 2_{in};c2_{in};\sigma \iota 1'_{in};\iota 2'_{out} \\ & = & \rho 2_{inter};\iota 2_{out};G2[c2_{in};\sigma \iota 1'_{in}]\end{array}$$

by the construction of $\Sigma 12_{out}$ as a pushout object, there exists a unique morphism

$$\tau : \Sigma 12_{out} \to G2(G1(\Sigma_{arg}[\sigma])[c2_{in};\sigma;\iota 1'_{out}])$$

such that

$$c1_{out};\tau = G1[c1_{in};\sigma];\iota 2'_{out} \qquad \text{and} \qquad c2_{out};\tau = G2[c2_{in};\sigma \iota 1'_{in}].$$

Then, since we have:

$$\begin{array}{rcl}\rho 1_{inter};c1_{shr};\iota 12_{in};\sigma;\iota 1'_{out};\iota 2'_{out} & = & \rho 1_{inter};\iota 1_{in};c1_{in};\sigma;\iota 1'_{out};\iota 2'_{out} \\ & = & \rho 1_{inter};\iota 1_{out};G1[c1_{in};\sigma];\iota 2'_{out} \\ & = & \rho 1_{inter};\iota 1_{out};c1_{out};\tau \\ & = & \rho 1_{inter};c1_{shr};\iota 12_{out};\tau\end{array}$$

and similarly

$$\rho 2_{inter};c2_{shr};\iota 12_{in};\sigma;\iota 1'_{out};\iota 2'_{out} = \rho 2_{inter};c2_{shr};\iota 12_{out};\tau$$

by the construction of $\Sigma 12_{shr}$ as a pushout object,

$$\iota 12_{in};\sigma;\iota 1'_{out};\iota 2'_{out} = \iota 12_{out};\tau.$$

Hence, by the construction of $G12(\Sigma_{arg}[\sigma])$ as a pushout object, there exists a unique morphism

$$\tau_{fin} : G12(\Sigma_{arg}[\sigma]) \to G2(G1(\Sigma_{arg}[\sigma])[c2_{in};\sigma;\iota 1'_{out}])$$

such that $\iota 12'_{out};\tau_{fin} = \iota 1'_{out};\iota 2'_{out}$ and $G12[\sigma];\tau_{fin} = \tau$.

Claim: $$G2_{gres}(G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}])\big|_{\tau_{fin}} = G12_{gres}(A[\sigma]).$$

In particular, this implies that $A\big|_{\sigma} \in Dom(G12)$.

To verify the claim, we check that the reducts of the left-hand side to Σ_{arg} and $\Sigma 12_{out}$ (via $\iota 12'_{out}$ and $G12[\sigma]$ respectively) are A and $G12_{bsem}(A\big|_{\sigma})$ (and that $G12_{bsem}(A\big|_{\sigma})$ is defined):

$$\begin{array}{rl}& (G2_{gres}(G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}])\big|_{\tau_{fin}})\big|_{\iota 12'_{out}} \\ = & G2_{gres}(G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}])\big|_{\iota 1'_{out};\iota 2'_{out}} \\ = & G1_{gres}(A[c1_{in};\sigma])\big|_{\iota 1'_{out}} \\ = & A.\end{array}$$

Then, since

$$\begin{array}{rl}& ((\,G2_{gres}(\,G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}]\,)\big|_{\tau_{fin}}\,)\big|_{G12[\sigma]}\,)\big|_{c2_{out}} \\ = & G2_{gres}(\,G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota 1'_{out}]\,)\big|_{G2[c2_{in};\sigma;\iota 1'_{out}]} \\ = & G2_{bsem}(\,G1_{gres}(A[c1_{in};\sigma]\,)\big|_{c2_{in};\sigma;\iota 1'_{out}} \\ = & G2_{bsem}(\,(A\big|_{\sigma})\big|_{c2_{in}}\,)\end{array}$$

and

$$
\begin{aligned}
(\,(\,G2_{gres}(\,G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota1'_{out}]\,)\big|_{\tau_{fin}}\,)\big|_{G12[\sigma]}\,)\big|_{c1_{out}} \\
&= G2_{gres}(\,G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota1'_{out}]\,)\big|_{G1[c1_{in};\sigma];\iota2'_{out}} \\
&= G1_{gres}(\,A[c1_{in};\sigma]\,)\big|_{G1[c1_{in};\sigma]} \\
&= G1_{bsem}(\,(A\big|_{\sigma})\big|_{c1_{in}}\,),
\end{aligned}
$$

we conclude that $A\big|_{\sigma} \in Dom(G12)$ and

$$(\,G2_{gres}(\,G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota1'_{out}]\,)\big|_{\tau_{fin}}\,)\big|_{G12[\sigma]} = G12_{bsem}(A\big|_{\sigma}),$$

which proves the claim.

Now, consider $\widehat{B} =_{def} B2\big|_{\tau_{fin}}$. We have

- $\widehat{B}\big|_{\iota12'_{out}} = B2\big|_{\iota12'_{out};\tau_{fin}} = B2\big|_{\iota1'_{out};\iota2'_{out}} = B$,
- $\widehat{B} \equiv_{sorts(\mathtt{Perv})} G12_{gres}(A[\sigma])$, since $B2 \equiv_{sorts(\mathtt{Perv})} G2_{gres}(G1_{gres}(A[c1_{in};\sigma])[c2_{in};\sigma;\iota1'_{out}])$, and
- $\widehat{B}\big|_{G12[\sigma]} \models SIG12_{out}$, since
  - $(\widehat{B}\big|_{G12[\sigma]})\big|_{c1_{out}} = B2\big|_{G1[c1_{in};\sigma];\iota2'_{out}} = B1\big|_{G1[c1_{in};\sigma]} \models SIG1_{out}$, and
  - $(\widehat{B}\big|_{G12[\sigma]})\big|_{c2_{out}} = B2\big|_{G2[c2_{in};\sigma;\iota1'_{out}]} \models SIG2_{out}$.

This completes the proof of the proposition. □

Theorem 5.5 *Consider an Extended ML functor F:*

$$\mathtt{functor}\ F(X : SIG_{in}) : SIG_{out}\ \mathtt{sharing}\ \textit{sharing-decl} = G0(G1(X), G2(X))$$

where $G0$, $G1$ and $G2$ are functors with headings:

$$
\begin{array}{l}
\mathtt{functor}\ G0(Y01 : SIG01_{in}, Y02 : SIG02_{in}\ \mathtt{sharing}\ \textit{sharing-constr}0) : SIG0_{out} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mathtt{sharing}\ \textit{sharing-decl}0 \\
\mathtt{functor}\ G1(Y1 : SIG1_{in}) : SIG1_{out}\ \mathtt{sharing}\ \textit{sharing-decl}1 \\
\mathtt{functor}\ G2(Y2 : SIG2_{in}) : SIG2_{out}\ \mathtt{sharing}\ \textit{sharing-decl}2
\end{array}
$$

Suppose that the definition of F is well-formed according to the Standard ML typechecking rules, which determine algebraic signature morphisms as described in Section 5.3. Suppose that the following conditions are satisfied:

1. (a) $SIG_{in} \models_{\Sigma1_{in}}^{sorts(\iota1_{in}(\rho'(\Sigma'_{shr})))\cup sorts(\iota1_{in}(\rho1_{inter}(\Sigma0_{com})))} SIG1_{in}$
   (b) $SIG_{in} \models_{\Sigma2_{in}}^{sorts(\iota2_{in}(\rho''(\Sigma''_{shr})))\cup sorts(\iota2_{in}(\rho2_{inter}(\Sigma0_{com})))} SIG2_{in}$
2. (a) $SIG1_{out} \models_{\Sigma01_{in}}^{sorts(\iota01_{in}(\rho0'(\Sigma'_{shr})))\cup sorts(\iota01_{com}(\Sigma0_{com}))} SIG01_{in}$
   (b) $SIG2_{out} \models_{\Sigma02_{in}}^{sorts(\iota02_{in}(\rho0''(\Sigma''_{shr})))\cup sorts(\iota02_{com}(\Sigma0_{com}))} SIG02_{in}$
3. $SIG0_{out} \models_{\Sigma_{out}}^{sorts(\Sigma_{shr})} SIG_{out}$

Then, if $G0$, $G1$ and $G2$ are universally correct then so is F.

Proof We show that the conditions above imply the corresponding conditions of Corollary 5.4. The third conditions are the same in both cases. As for the other two:

1. $SIG_{in} \models^{sorts(\iota 12_{in}(\rho(\Sigma_{shr})))}_{\Sigma 12_{in}} SIG12_{in}$:

   Consider any $A \in Mod[SIG_{in}]$. We have to construct a $\Sigma 12_{in}$-algebra B such that

   $$A\big|_{\tau_{in}} \equiv_{sorts(\iota 12_{in}(\rho(\Sigma_{shr})))} B \qquad \text{and} \qquad B \models SIG12_{in}.$$

   By assumption 1a, there exists $B1 \in Mod[SIG1_{in}]$ such that

   $$A\big|_{\tau 1_{in}} \equiv_{sorts(\iota 1_{in}(\rho'(\Sigma'_{shr}))) \cup sorts(\iota 1_{in}(\rho 1_{inter}(\Sigma 0_{com})))} B1.$$

   Similarly, by assumption 1b, there exists $B2 \in Mod[SIG2_{in}]$ such that

   $$A\big|_{\tau 2_{in}} \equiv_{sorts(\iota 2_{in}(\rho''(\Sigma''_{shr}))) \cup sorts(\iota 2_{in}(\rho 2_{inter}(\Sigma 0_{com})))} B2.$$

   It follows that

   $$B1\big|_{\rho 1_{inter};\iota 1_{in}} = A\big|_{\rho 1_{inter};\iota 1_{in};\tau 1_{in}} = A\big|_{\rho 2_{inter};\iota 2_{in};\tau 2_{in}} = B2\big|_{\rho 2_{inter};\iota 2_{in}}.$$

   Hence, we can construct the amalgamated union of $B1$ and $B2$, i.e. the unique $\Sigma 12_{in}$-algebra B such that $B\big|_{c1_{in}} = B1$ and $B\big|_{c2_{in}} = B2$. By the definition of $SIG12_{in}$, $B \models SIG12_{in}$. Moreover, since $A\big|_{\tau_{in}}$ is the amalgamated union of $A\big|_{\tau 1_{in}}$ and $A\big|_{\tau 2_{in}}$, by Lemma A.1 we have

   $$A\big|_{\tau_{in}} \equiv_{sorts(c1_{in}(\iota 1_{in}(\rho'(\Sigma'_{shr})))) \cup sorts(c2_{in}(\iota 2_{in}(\rho''(\Sigma''_{shr})))) \cup \ldots} B$$

   which implies

   $$A\big|_{\tau_{in}} \equiv_{sorts(\iota 12_{in}(\rho(\Sigma_{shr})))} B.$$

2. $SIG12_{out} \models^{sorts(\iota 0(\rho 0(\Sigma_{shr})))}_{\Sigma 0_{in}} SIG0_{in}$:

   Consider $A \in Mod[SIG12_{out}]$. We have to construct a $\Sigma 0_{in}$-algebra B such that

   $$A\big|_{\tau_{inter}} \equiv_{sorts(\iota 0_{in}(\rho 0(\Sigma_{shr})))} B \qquad \text{and} \qquad B \models SIG0_{in}.$$

   By the definition of $SIG12_{out}$, $A\big|_{c1_{out}} \models SIG1_{out}$, and so by assumption 2a, there exists $B1 \in Mod[SIG01_{in}]$ such that $A\big|_{\tau 1_{inter};c1_{out}} \equiv_{sorts(\iota 01_{in}(\rho 0'(\Sigma'_{shr}))) \cup sorts(\iota 01_{com}(\Sigma 0_{com}))} B1$. Similarly, by assumption 2b, there exists $B2 \in Mod[SIG02_{in}]$ such that

   $$A\big|_{\tau 2_{inter};c2_{out}} \equiv_{sorts(\iota 02_{in}(\rho 0''(\Sigma''_{shr}))) \cup sorts(\iota 02_{com}(\Sigma 0_{com}))} B2.$$

   As in the previous case, we can construct the amalgamated union $B \in Alg(\Sigma 0_{in})$ of $B1$ and $B2$ and then show that $B \models SIG0_{in}$ and $A\big|_{\tau_{inter}} \equiv_{sorts(\iota 0(\rho 0(\Sigma_{shr})))} B$.

$\square$