J. Functional Programming 4 (3): 285–335, July 1994 © 1994 Cambridge University Press

Principal signatures for higher-order program modules

MADS TOFTE

Department of Computer Science, University of Copenhagen, Denmark

Abstract

In this paper we present a language for programming with higher-order modules.[†] The language HML is based on Standard ML in that it provides structures, signatures and functors. In HML, functors can be declared inside structures and specified inside signatures; this is not possible in Standard ML. We present an operational semantics for the static semantics of HML signature expressions, with particular emphasis on the handling of sharing. As a justification for the semantics, we prove a theorem about the existence of principal signatures. This result is closely related to the existence of principal type schemes for functional programming languages with polymorphism.

Capsule review

One of the more successful and innovative features of Standard ML is its approach to modular programming. Interfaces, or *signatures*, describe the components of a program unit through type declarations and *sharing specifications*. Implementations, or *structures*, provide the actual code of a program unit, and are checked for conformance with their signatures. Implementations are combined using *functors*, functions mapping structures to structures. In Standard ML, functors are first-order; functors may not be passed as arguments or returned as results. This restriction has proved to be limitative in practice.

Standard ML is notable for having a rigorously defined static and dynamic semantics. The dynamic semantics defines the rules of evaluation, and is relatively standard. The static semantics defines a set of context-sensitive conditions that well-formed programs are required to satisfy, including the familiar typing constraints of the core language and analogous, but more complex, constraints at the module level. A critical property of the static semantics is the existence of *principal signatures* which summarize the compile-time properties of a module. This paper is concerned with extending this property to an extension of Standard ML with higher-order modules.

1 Introduction

Working on large programs involves manipulating large program units as well as working on the details of individual units. Such program units are sometimes called

[†] This is a revised and expanded version of a paper with the same title, presented at the *19th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, January 1992.

modules, especially if the programming language in question allows the programmer to name units and combine them in a controlled fashion.

Since managing large collections of program modules is a central task in practical programming, it is not surprising that programming languages are going through a development of more and more powerful language constructs for programming with modules. In ADA there is a basic form of module, called a *package*; module interfaces can be written in programs and are called *package interfaces*. Finally, a *generic package* is a package which has been parameterized on a type, making it possible to separate the implementation of the type from its use.

Standard ML has similar concepts, namely *structures*, *signatures* and *functors*, due to MacQueen (1984). A structure can declare datatypes and functions that operate on these types. A signature can specify the names of types and functions, but it does not necessarily tie these specifications to actual datatypes and functions.

Perhaps the single most important construct in the ML Modules language is the concept of *functor*, the Standard ML notion of parametric module. A functor can be thought of as a map from structures to structures. Consider the following Standard ML functor:

```
functor F(X:
            sig
              type t               parameter signature
              val a:t
            end
         ) =
   struct
     type pair = X.t * X.t         functor body
     val b:pair = (X.a, X.a)
   end
```

As indicated by the boxes, this functor declaration is of the general form

functor *funid*(*strid* : *sigexp*) ⟨: *sigexp′*⟩=*strexp*

Here *funid* is a functor identifier, *strid* is a structure identifier (the *formal parameter*) *sigexp* is a *signature expression* (the *parameter signature*) and *strexp* is a *structure expression*, called the *body* of the functor. When present, *sigexp′* is called the *result signature*. (Throughout this paper, the angle brackets ⟨⟩ enclose optional phrases.) The above functor `F` takes as argument a structure containing a type `t` and a value `a` and produces a structure containing a type `pair` and a value `b`.

Inside the functor body, one can refer to the components of the formal parameter — hence `X.t` and `X.a` in the above example. However, the references to the formal parameter *strid* must be valid assuming only what the parameter signature reveals about *strid*. Thus writing `X.z` in the body of `F` would be illegal, for `X` is not specified to have a `z` component. More interestingly, writing `(X.a)+1` in the body would be illegal, since `X.a` is of type `t` and `t` has not been specified to be the type of integers.

Practical advantages of programming with functors are:

1. One can write a program piece *P*, without first having to decide on the

implementation of the types and operations P depends on. P is simply made into the body of a functor F. As P is written, types and operations that are needed for P, but do not belong with P, are made into parameters of F.

2. A compiler can type-check F, even before the actual argument, to which F will eventually be applied, is written. Once F is type correct, one can usually forget about the details of the body of F and concentrate on the argument and result signatures.
3. When F eventually is applied to an actual argument structure S, it is automatically checked that S matches the parameter signature. If S does not match the argument signature (because one has forgotten to define some function in S, say) then an error message is produced. This eases the burden of keeping track of what has yet to be implemented.

It is perhaps not surprising that functors turn out to be so useful, for they are simply the modules variant of functions and it is well known that functions are useful in programming. Indeed, in functional programming languages, one insists that functions *are* values and as such can be stored in data structures, passed as arguments to functions, returned from functions, and so on. Similar generality is clearly in demand for parameterized modules. If one wants to write a piece of code which uses a functor `F`, but has no desire to write `F` just yet, the natural thing would be to write a functor `G`, parameterized on `F`:

```
functor G(X:sig ···functor F: ···  end) =
struct ···  X.F ···   end
```

Here `G` is an example of a *higher-order functor*, by which we mean a functor which is parameterized on a functor or returns a functor as result.

However, higher-order functors are not available in Standard ML. In this paper we present a skeletal language, HML, which is based on Standard ML, but admits higher-order functors. HML is not the first programming language with higher-order modules. Harper, Mitchell and Moggi (1990) propose a very elegant type-theoretic module concept which allows higher-order modules. Unfortunately, their approach does not address generativity, sharing and multiple structure views, all of which are important in Standard ML. In HML we do deal with these concepts and the resulting language is largely compatible with the first-order modules of Standard ML.

In this paper we do not present a complete semantics for HML. Dynamic semantics is not described at all and we omit the semantics of functor application and signature matching from the static semantics. What is left is the static semantics of signature expressions containing functor specifications. Such signatures are far from trivial to deal with, however. We shall justify our particular semantics for signature expressions by proving that if a signature expression is legal according to the semantics, then it has a so-called *principal signature*, the modules equivalent of principal type scheme (Milner, 1978; Damas and Milner, 1982).

In section 2 we give the grammar for HML and explain the language informally. In section 3 we define the static semantics of signature expressions and specifications using 'natural semantics' (or 'relational semantics'). In section 4 we discuss the connection between sharing constraints and unification. In section 5 we prove a theorem called the realization theorem; in section 6 we prove the main result about principal signatures using a signature inference algorithm W which is presented at the same time. Finally, section 7 presents our conclusions.

2 A skeletal language

In this section we give an informal presentation of HML.

2.1 Grammar

We assume three disjoint, denumerably infinite identifier classes:

strid	$\in$ StrId	structure identifier	long
sigid	$\in$ SigId	signature identifier	
funid	$\in$ FunId	functor identifier	long

For each class X marked 'long' there is a class longX of *long identifiers*; if x ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

longx	::=	x	identifier
		$strid_1.\cdots.strid_n.x$	qualified identifier ($n \geq 1$)

The phrase classes of HML are:

strexp	$\in$	StrExp	structure expression
funexp	$\in$	FunExp	functor expression
atstrdec	$\in$	AtStrDec	atomic structure-level declaration
strdec	$\in$	StrDec	structure-level declaration
sigexp	$\in$	SigExp	signature expression
funsigexp	$\in$	FunSigExp	functor signature expression
atspec	$\in$	AtSpec	atomic specification
spec	$\in$	Spec	specification
shareq	$\in$	SharEq	sharing equation
atprogram	$\in$	AtProgram	atomic program
program	$\in$	Program	program

For expository reasons, we present a grammar both for signatures (Fig. 1) and for structures and functors (Fig. 2) although it is only the language of Fig. 1 that we study in detail in this paper. Since Fig. 1 does not refer to the phrase classes defined in Fig. 2, the theorems we prove do not rely on a particular semantics for the constructs of Fig. 2. We write `structure` *strid* : *sigexp* = *strexp* for `structure` *strid* = *strexp* : *sigexp*. Similarly, we write

$$\texttt{functor}\ \mathit{funid}\,(\mathit{strid} : \mathit{sigexp})\langle : \mathit{sigexp}'\rangle\ =\ \mathit{strexp}$$

for

$$\texttt{functor}\ \mathit{funid}\ =\ \texttt{func}\ \mathit{strid} : \mathit{sigexp}\ =\ \mathit{strexp}\langle : \mathit{sigexp}'\rangle$$

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

sigexp	::=	`sig` *spec* `end`	basic
		sigid	signature identifier
		sigexp `is` *strid* `sharing` *shareq*	sharing qualification
funsigexp	::=	(*strid* : $sigexp_1$) $sigexp_2$	functor signature
atspec	::=	`structure` *strid* : *sigexp*	structure
		`functor` *funid* : *funsigexp*	functor
		`sharing` *shareq*	sharing
		`local` $spec_1$ `in` $spec_2$ `end`	local
spec	::=		empty
		atspec ⟨;⟩ *spec*	sequence
shareq	::=	$longstrid_1$ = $longstrid_2$	sharing equation

Fig. 1. Grammar for signatures.

strexp	::=	`struct` *strdec* `end`	generative
		longstrid	structure identifier
		funexp(*strexp*)	functor application
		strexp : *sigexp*	signature constraint
funexp	::=	`func` *strid* : *sigexp* => *strexp*	functor
		longfunid	functor identifier
		(*funexp*)	
atstrdec	::=	`structure` *strid* = *strexp*	structure
		`functor` *funid* = *funexp*	functor
strdec	::=		empty
		atstrdec ⟨;⟩ *strdec*	sequence
atprogram	::=	*atstrdec*	
		`signature` *sigid* = *sigexp*	signature declaration
program	::=	*atprogram* ; ⟨*program*⟩	program

Fig. 2. Grammar for structures and functors.

Thus `func` is the 'λ-abstraction of HML'. The scope of *strid* is *strexp* (and *sigexp'*, if present). The `func` phrase form extends as far right as possible; with this convention, the grammar is unambiguous.

In a functor signature expression (*strid* : $sigexp_1$)$sigexp_2$ the scope of *strid* is $sigexp_2$. In examples, we take the liberty to extend the structure-level declarations with declarations of values and types; similarly, we allow specifications of values and types.

Example 2.1 Figure 3 shows an example of programming with first-order functors. First a signature `MONOID` is declared (a). Then two monoids `Int` and `String` are

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

```
signature MONOID =
sig
   type t
   val e: t
   val plus: t * t -> t

end;
```
(a)

```
structure Int: MONOID=
struct
   type t = int
   val e = 0
   fun plus(x,y):int = x+y
end;
```
(b)

```
structure String: MONOID=
struct
   type t = string
   val e = ""
   fun plus(s1,s2)=s1^s2
end
```
(c)

```
signature MPAIR =
sig
   structure M: MONOID
   structure N: MONOID
end;
```
(d)

```
functor Prod(X: MPAIR):MONOID =
struct
   type t = X.M.t * X.N.t
   val e = (X.M.e, X.N.e)
   fun plus((x1,x2),(y1,y2))=
     (X.M.plus(x1,y1),
      X.N.plus(x2,y2))

end;
```
(e)

```
structure TitleAndAge =
Prod(struct
        structure M = String
        structure N = Int
      end);
```
(f)

```
··· TitleAndAge.plus(
      student, ("M.Sc.",4))
```
(g)

Fig. 3. Examples of modules.

declared — see (b) and (c). Note that (a) specifies `t`, `e` and `plus` without saying what they are. `Int` and `String` give different implementations of `MONOID`; for example, `plus` is addition of integers in `Int` but concatenation of strings in `String`. The signature constraints ': `MONOID`' in (b) and (c) serve to check that `Int` and `Real` really do match the `MONOID` signature. The next signature (d) can be matched by any structure that has substructures `M` and `N` both of which must match `MONOID`. Functor `Prod` (e) can create a monoid, namely the product of `M` and `N`, for every structure `X` that matches `MPAIR`. At (f), functor `Prod` is applied. Notice that the actual argument is a structure which matches `MPAIR`. This application yields a structure, called `TitleAndAge`. The expression at (g) will graduate `student` by appending the string `"M.Sc."` to the title and 4 years to the age. □

Example 2.2 Figure 4 illustrates the use of a higher-order functor, `Square`, which is declared at (h). Here '`functor Product: (X:MPAIR)MONOID`' is an example of a functor specification. The body of `Square` is just the application of `Product`. Notice that `Square` is *closed*, i.e. it contains no free identifiers except signature identifiers; thus it can be compiled before functor `Prod` (Figure 3(e)) is written. Once `Prod` is

```
functor Square(X:
sig
   functor Product: (X:MPAIR)MONOID
   structure Y: MONOID
end): MONOID =
  X.Product(
   struct
     structure M = X.Y
     structure N = X.Y
   end);
```
(h)

```
structure Plane =
   Square(
     struct
       functor Product = Prod
       structure Y= Int
     end);
```
(i)

```
···  Plane.plus((1,3),(2,5))
```
(j)

Fig. 4. Use of higher-order functor.

```
structure A =
struct
   structure B = struct end
   functor F(X: sig end)= struct structure C = X end
   structure D = struct end
end;

signature SIG =
sig
   structure B: sig end
   functor F: (X: MONOID) sig end
end;

structure A':SIG = A;
```

Fig. 5. A signature constraint can lead to multiple views of a structure.

declared, we can apply `Square` to a structure containing `Prod`, see (i). This gives a monoid, `Plane`, of integer pairs. At (j), we see a use of the `Plane` structure. □

The reader will have noticed that it is sometimes a bit cumbersome to wrap up functor arguments in structures. In examples, we shall occationally use the following alternative phrase forms:

strexp	::=	*funexp*`(`*strdec*`)`
atstrdec	::=	`functor` *funid*`(`*spec*`)`⟨`:`*sigexp′*⟩`=` *strexp*
funsigexp	::=	`(`*spec*`)`*sigexp*

2.2 Multiple views of structures

In matching a structure S against a signature Σ, the structure must have at least the components specified by Σ. Moreover, the functor components of S must be at least as general as specified. Consider the declarations in Fig. 5. Here `A` has the components required by `SIG` plus an additional `D` structure. The declared functor `F` requires no more of its argument than the specified functor `F` and it produces at

```
sig                                  sig
   structure M: MONOID                  structure M: MONOID
   structure N: MONOID                  structure N: MONOID
   sharing type M.t = N.t               sharing M = N
end                                  end
            (a)                                  (b)
```

Fig. 6. Sharing specifications.

least as much as the specified `F` — note the contravariance in the argument position. Thus `A` matches `SIG`.

The structure `A'` has only the components specified by `SIG`. Moreover, the functor `A'.F` is treated as having the functor signature specified in `SIG`. Thus it can only be applied to structures that match the *specified* parameter signature (i.e. `MONOID`) and the result of applying `A'.F` will be constrained by the *specified* result signature.

In the example, `A` and `A'` can be thought of as different *views* of the same original structure. In particular, `A.F` and `A'.F` are 'really' the same functor, just seen through two different views. Similarly, if `A` had been able to declare a datatype or a value which was also specified in `SIG` then `A'` would contain that same datatype or value, although perhaps with a different view.

The Standard ML modules system makes it possible to determine statically whether two structures are (perhaps different) views of the same original structure. The basic rule is that an occurrence of a generative structure expression (i.e. an expression of the form `struct` *strdec* `end`) generates one fresh structure, provided the occurrence is not inside a functor body; otherwise, the occurrence generates a fresh structure each time the closest enclosing functor is called. As an example, the total number of structures generated by Fig. 3 is 4; if the declarations in Fig. 4 are subsequently executed, an additional three structures are generated.

2.3 Sharing

A structure specification specifies a view of a structure, as opposed to a particular structure. It is sometimes necessary to specify that two specified structures must be (perhaps different) views of the same structure. This is particularly true of structure specifications in parameter signatures. For example, consider the following declaration:

```
functor F(X: sig structure M: MONOID; structure N: MONOID end) =
struct ···X.M.plus(X.M.e,X.N.e) ···  end
```

This functor is illegal, for good reasons. `X.M.e` and `X.N.e` have types `X.M.t` and `X.N.t`, respectively, so the `plus` operation does not make sense unless we have `X.M.t = X.N.t`. Since the body of the functor must be valid assuming only what the parameter signature specifies, we need to specify that the two types must be two views of the same (unknown) type. In ML, this is done with a *type sharing* specification. For example, one could replace the parameter signature above with the

```
structure A = struct end;
signature SIG =
sig
   structure A1: sig end
   sharing A1 = A
end
```
(a)

```
functor F:(X: sig end)
   sig
     structure Y: sig end
     sharing Y = X
   end
```
(b)

Fig. 7. Sharing: (a) with declared structure and (b) between argument and result in a functor specification.

signature expression in Fig. 6(a). One can also specify sharing of entire structures, see Fig. 6(b). A specification that two structures S_1 and S_2 share implicitly is a specification that identically named structures and types visible in both structures share as well. For example, every structure which matches Fig. 6(b) also matches Fig. 6(a), but the converse is not true.

In HML we do not have type sharing (as we do not have types). However, besides the structure sharing form, HML makes it possible to qualify every signature expression by a sharing equation. For example, the signature in Fig. 6(b) can be written thus:

```
MPAIR is Y sharing Y.M = Y.N
```

in the scope of the declaration of `MPAIR` in Fig. 3. For all signature expressions of the form *sigexp* `is` *strid* `sharing` *shareq* the scope of the structure identifier *strid* is just the sharing equation *shareq*. Since this language construct serves to qualify a signature expression by a sharing equation, we call it a *sharing qualification.*

A structure identifier mentioned in a sharing equation must be in scope at the place the sharing equation occurs. Figure 6(b) specifies sharing between two specified structures. Figure 7(a) specifies sharing with a *declared* structure. Figure 7(b) specifies sharing between the *argument and the result* of a functor. As a final example, here is the 'identity' functor and, below it, the most accurate specification of it:

```
functor Id(X: sig end) = X
functor Id:(X: sig end) sig end is Y sharing Y=X
```

This functor can be applied to any structure S and returns a structure which shares with S but has no visible components! There is no way of declaring or specifying a functor which can be applied to all structures and returns its argument unchanged.

2.4 Local and overlapping specifications

Like Standard ML, HML has `local` specifications. Local specifications can be used to express fairly advanced sharing constraints. One of the referees provided the following nice example:

```
functor H(
  local structure A: sig end
  in    functor F: () sig structure B: sig end sharing B = A end
```

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

```
  end;
  functor G:(structure X: sig end
             structure Y: sig end
             sharing X=Y)sig end
  ): sig structure Z: sig end end =
struct
  structure X1 = F()
  structure Y1 = F()
  structure Z = G(structure X=X1.B; structure Y=Y1.B)
end;
```

Here `local` makes it possible to specify sharing between the results of different applications of `F`. Thus the application of `G` is valid and would not be valid if the sharing specification in the specification of `F` were removed. Without the `local` specification, there seems to be no way of achieving this effect without making `A` a parameter of `H`.

The present semantics for signature expressions admits local specifications. It even allows overlapping sequential specifications, by which we mean specifications of the form *atspec*⟨;⟩*spec*, where some structure- or functor-identifier is specified by both *atspec* and *spec*.

There are reasons why one might want to restrict the use of local and overlapping specifications in full HML. In particular, the concept of matching a structure against a signature appears to become considerably more complex, if signatures like the parameter signature of `H` are allowed. Local and overlapping specifications also complicate the semantics of signatures somewhat but, as we shall see, principal signatures can be inferred even so.

3 A static semantics of HML signatures

In this section we present a relational semantics for signature expressions. We define principality and state the principality theorem.

3.1 Notation

When A and B are sets $\mathrm{Fin}(A)$ denotes the set of finite subsets of A, and $A \xrightarrow{\mathrm{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map, f, are denoted $\mathrm{Dom}(f)$ and $\mathrm{Ran}(f)$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$. When f and g are finite maps the map $f + g$, called f *modified by* g, is the finite map with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values

$$(f + g)(a) = \text{if } a \in \mathrm{Dom}(g) \text{ then } g(a) \text{ else } f(a)$$

The restriction of f to A is written $f \downarrow A$. When A and B are sets $A \uplus B$ denotes the disjoint union of A and B. The above definitions are largely taken directly from the Definition of Standard ML (Milner *et al.*, 1990).

m	$\in$	StrName	structure name
N	$\in$	NameSet = Fin(StrName)	name set
G	$\in$	SigEnv = SigId $\xrightarrow{\text{fin}}$ Sig	signature environment
FE	$\in$	FunEnv = FunId $\xrightarrow{\text{fin}}$ FunSig	functor environment
SE	$\in$	StrEnv = StrId $\xrightarrow{\text{fin}}$ Str	structure environment
S or (m, E)	$\in$	Str = StrName $\times$ Env	structure
E or (FE, SE)	$\in$	Env = FunEnv $\times$ StrEnv	environment
Σ or $(N)S$	$\in$	Sig = NameSet $\times$ Str	signature
Φ or $(N)(S,(N')S')$	$\in$	FunSig = NameSet $\times$ (Str $\times$ Sig)	functor signature
B or N, G, E	$\in$	Basis = NameSet $\times$ SigEnv $\times$ Env	basis
A	$\in$	Asmb = EmptyAsmb $\uplus$ (Str $\times$ Asmb)	assembly
		EmptyAsmb = $\{\epsilon\}$	

Fig. 8. Semantic objects.

3.2 Assemblies and structures

The term *elaboration* is used for that part of execution which pertains to the static semantics. The statement that some phrase *phrase* elaborates to a result D, starting from C, will be written $C \vdash phrase \Rightarrow D$. Here C and D are so-called *semantic objects.* The semantic objects for HML are defined by the set equations in Fig. 8. We use $\mathcal{O}$ to range over semantic objects. To explain the meaning of the semantic objects, in broard terms at least, let us start by considering the elaboration of structure expressions. The statement

$$A, B \vdash strexp \Rightarrow S, A' \tag{1}$$

is read: *in assembly A and basis B the structure expression strexp elaborates to structure S and a (perhaps expanded) assembly A'*. The basis, B, is used for looking up the meaning of the free identifiers of *strexp* — see Fig. 8. The structure S can be thought of as the static value of *strexp*. The assembly, A, is essentially a list $[S_1, \ldots, S_n]$ of structures — see Fig. 8. It acts as a static structure store. When a new structure is created, it is put into the assembly. It is possible to have restricted views of structures after they have originally been created and such restricted views can exist in places where the original structure is not in scope. In signature expressions sharing specifications can even specify different views of some purely hypothetical structure. The technical purpose of the assembly is to serve as a common frame of reference for different restricted views of the same structure.

We assume a denumerably infinite set StrName of *structure names.* We use m to range over structure names and N to range over finite sets of names. A structure name can be thought of as a unique name (or stamp) of the structure in question; name binding by nameset prefixes is used for delimiting the scope of uniqueness, as detailed below. A *structure* S is a pair (m, E), where E is an environment and m is *the name of* S. Two structures $S_1 = (m_1, E_1)$ and $S_2 = (m_2, E_2)$ *share* if they have the same name, i.e. if $m_1 = m_2$.

An *environment* is a pair $E = (FE, SE)$, where FE is a *functor environment* and SE is a *structure environment.* FE maps functor identifiers to functor signatures while SE maps structure identifiers to structures.

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

$$
\begin{aligned}
\text{names}((m, E)) &= \{m\} \cup \text{names}(E) \\
\text{names}\big((FE, SE)\big) &= \text{names}(FE) \cup \text{names}(SE) \\
\text{names}(G) &= \cup\{\text{names}(G(\mathit{sigid})) \mid \mathit{sigid} \in \text{Dom}(G)\} \\
\text{names}(FE) &= \cup\{\text{names}(FE(\mathit{funid})) \mid \mathit{funid} \in \text{Dom}(FE)\} \\
\text{names}\big(SE\big) &= \cup\{\text{names}(SE(\mathit{strid})) \mid \mathit{strid} \in \text{Dom}(SE)\} \\
\text{names}\big((N)S\big) &= \text{names}(S) \setminus N \\
\text{names}\big((N)(S, (N')S')\big) &= \big(\text{names}(S) \cup \text{names}\big((N')S'\big)\big) \setminus N \\
\text{names}((N, G, E)) &= N \cup \text{names}(G) \cup \text{names}(E) \\
\text{names}(A) &= \cup\{\text{names}(S) \mid S \text{ is an element of the list } A\}
\end{aligned}
$$

Fig. 9. Names that occur free in objects.

We often need to select parts of semantic objects — for example the name of a structure. In such cases we rely on variable names to indicate which part is selected. For instance 'm of S' means 'the structure name of S'. Moreover, when a semantic object contains a finite map we shall 'apply' the object to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance $S(\mathit{strid})$ means (SE of (E of S))(strid). Furthermore, we use id to range over StrId$\cup$FunId and we write $\mathit{id} \in \text{Dom}(E)$ to mean '$\mathit{id} \in$ StrId and $\mathit{id} \in \text{Dom}(SE \text{ of } E)$ or $\mathit{id} \in$ FunId and $\mathit{id} \in \text{Dom}(FE \text{ of } E)$'.

Modification extends to environments: $E + E' = (FE \text{ of } E + FE \text{ of } E', SE \text{ of } E + SE \text{ of } E')$. Furthermore, it extends to bases, if we interpret $+$ on name sets as set union. Hence $(N, G, E) + N_1 = (N \cup N_1, G, E)$.

We shall often tacitly regard structure environments (or functor environments) as environments. An empty structure will often be written $(m, \{\})$, which means $(m, (\{\}, \{\}))$.

A *nameset prefix* (N) in a signature or a functor signature binds names. In a signature $(N)S$, the scope of the binding of the names in N is S. In a functor signature $(N)(S, (N')S')$, the scope of (N) is $(S, (N')S')$ (and the scope of (N') is S'). Signatures and functor signatures will be explained in sections 3.3 and section 3.6, respectively.

Nameset prefixes give rise to the notions of free and bound occurrences of names, in the usual way. For any semantic object $\mathcal{O}$, the set of names that occur *free* in $\mathcal{O}$, written names($\mathcal{O}$), is defined by the equations in Fig. 9. Semantic objects that can be obtained from each other by renaming of bound names are considered equal.

The *proper substructures* of $S = (m, (FE, SE))$ are the members of the range of SE and their proper substructures. The *substructures* of S are S itself and its proper substructures.

For any semantic object $\mathcal{O}$ the semantic objects *occurring inside* $\mathcal{O}$ are the objects from which it is built, according to Fig. 8, and all the objects that occur inside them. The objects *occurring in* $\mathcal{O}$ are $\mathcal{O}$ and all the objects that occur inside $\mathcal{O}$. For instance, the structure S' occurs in the signature $(N')S'$, which in turn occurs in the functor signature $(N)(S,(N')S')$. Notice that not every structure which occurs in a structure S is necessarily a substructure of S. For example, S_1 and S_2 occur in the structure $S = (m, (\{f \mapsto (N_1)(S_1,(N_2)S_2)\}, \{\}))$, and so do all the structures that occur in S_1 and S_2, even though S has no proper substructures.

A structure (m, E) occurs *free* in A if (m, E) occurs in A at a position where the m is free in A.

Sharing is hereditary from structures to substructures. Formally, we define:

Definition 1 (Consistency)
A semantic object $\mathcal{O}$ is said to be *consistent* if (after changing bound names to make all nameset prefixes in $\mathcal{O}$ disjoint) for all S_1 and S_2 occurring in $\mathcal{O}$ and for every *strid*, if $m \text{ of } S_1 = m \text{ of } S_2$ and $S_1(strid)$ and $S_2(strid)$ exist, then $m \text{ of } S_1(strid) = m \text{ of } S_2(strid)$.

Notice that consistency does not impose a constraint on common functor components of S_1 and S_2. Thus the two structures `A` and `A'` of Fig. 5 are consistent. Consistency applies to specified structures as well as declared structures. Thus the following signature expression is legal:

```
sig
  structure A1: sig functor F: (X: sig end)sig end  end
  structure A2: sig functor F: (X: sig structure B: sig end end)
                                 sig structure B: sig end end
                  end
  sharing A1 = A2
end
```

Any real functor `F` which matches both the above specifications will have to be applicable to any structure and will then have to produce a structure with a `B` substructure. However, no attempt is made to synthesize this information from the two specifications.

It is sometimes helpful to think of an assembly A as a directed edge- and node-labelled graph. There is one node in the graph for every structure name $m \in \text{names}(A)$; in addition, there is a special node labelled `functor`. Furthermore, whenever $(m, E) = (m, (FE, SE))$ occurs free in A and $(m', E') = SE(strid)$, for some *strid*, there is precisely one edge labelled *strid* going from the node labelled m to the node labelled m'. Also, for all *funid* in the domain of FE, there is an edge labelled *funid* from the node labelled m to the node labelled `functor`. Informally, we refer to this graph as Graph(A).

The reason why there are no edges emanating from the `functor` node is that the assembly is used as a consistent frame of reference concerning sharing, but there is no way of specifying sharing of functors.

Corresponding to the notion of a graph being a subgraph of another, we have the following definition:

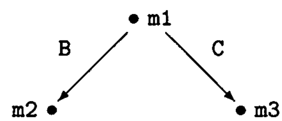
Definition 2 (Cover)
Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be semantic objects and N a name set. We say that $\mathcal{O}_2$ *covers* $\mathcal{O}_1$ *on* N if $N \cap \text{names}(\mathcal{O}_1) \subseteq \text{names}(\mathcal{O}_2)$ and also for all (m, E_1) and for all $id \in \text{Dom}\, E_1$ if (m, E_1) occurs free in $\mathcal{O}_1$ and $m \in N$, then there exists an E_2 such that (m, E_2) occurs

```
structure A =                    signature SIG =
struct                           sig
  structure B= struct end          structure B: sig end
  structure C= struct end        end;
end;                             structure A':SIG = A;
```

Fig. 10. A simple example of coercive signature matching.

$(\mathtt{m1}, \{\mathtt{B} \mapsto (\mathtt{m2}, \{\}), \mathtt{C} \mapsto (\mathtt{m3}, \{\})\})$

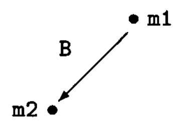
(a)



(b)

$(\mathtt{m1}, \{\mathtt{B} \mapsto (\mathtt{m2}, \{\})\})$

(c)



(d)

Fig. 11. The structure `A` of Fig. 10 elaborates to (a), pictured at (b), whereas `A'` elaborates to (c), pictured at (d).

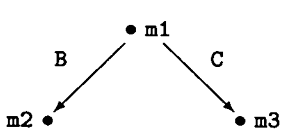
free in $\mathcal{O}_2$ and $id \in \mathrm{Dom}\, E_2$. We say that $\mathcal{O}_2$ *covers* $\mathcal{O}_1$ if $\mathcal{O}_2$ covers $\mathcal{O}_1$ on names($\mathcal{O}_1$). We say that $\mathcal{O}_2$ is a *conservative* cover of $\mathcal{O}_1$ if $\mathcal{O}_2$ covers $\mathcal{O}_1$ and $\mathcal{O}_1$ covers $\mathcal{O}_2$ on names($\mathcal{O}_1$).

To take an example, if $A, B \vdash strexp \Rightarrow S, A'$ and A covers B then A' covers A and S. Moverover, A' will be a conservative cover of A, for once a structure is generated, there is no way of adding components to it.

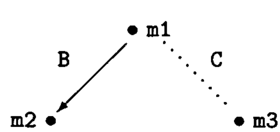
It is also useful to think of structures as edge- and node-labelled trees. For example, consider the declarations in Fig. 10. The structures they elaborate to are shown in Fig. 11. Note that trees can be labelled by the same structure name and yet have a different 'shape'.

Another approach, due to Aponte (1992; 1993), is to represent a cut-down view by decorating some of the components of the original as inaccessible, as indicated by the dashed line in Fig. 12(b).

The technique used in the present paper (using an assembly and requiring consistency and cover) and the technique used by Aponte (including invisble components in structures) both serve to ensure that consistency is preserved during elaboration.



(a)



(b)

Fig. 12. Different views of structures can be represented as trees that have the same shape but differ in what components they make visible.

In the former case, a 'global' data structure is used; in the latter case, local information in structures is used, together with the invariant that if two structures share, then they must have the same shape.

3.3 Signatures

Elaboration of a signature expression takes the following form:

$$A, B \vdash sigexp \Rightarrow \Sigma \tag{2}$$

where A is an assembly, B is a basis and Σ is a *signature*, the static value of *sigexp*. Note that the elaboration does not *produce* an assembly. This is because signature expressions and specifications have no way of generating new structures. On the contrary, Σ is a 'generic' (or *flexible*) structure which perhaps can be matched by many 'real' (or *rigid*) structures. Harper, Milner and Tofte (1987) make this distinction by defining a signature Σ to be an object of the form $(N)S$ where S is a structure and (N) is a nameset prefix. Whenever an actual structure, S', is matched against $(N)S$, one can instantiate the bound names to corresponding names in S'. A signature $(N)S$ is *closed*, if it contains no free names. An HML signature expression elaborates to a closed signature, unless it specifies sharing with a structure which is declared or specified outside the signature expression.

In ModL (Harper *et al.*, 1987), the first-order version of HML, there is a close correspondance between the type discipline for modules and Milner's type discipline for functional languages (Milner, 1987), namely:

Modules Language	*Functional Language*
structure, S	type, τ
signature, $\Sigma = (N)S$	type scheme, $\sigma = \forall\alpha_1 \cdots \alpha_n.\tau$

However, HML structures can contain functor signatures, which in turn contain signatures, so there is nested quantification in HML signatures. The algorithm in section 6 finds principal signatures for all legal signature expressions even so.

To have an easy way of distinguishing the substructures of a structure from the other structures that can occur in it, we define a function 'skel' (for skeleton) as follows:

$$\begin{aligned}
\mathrm{skel}(m, E) &= (m, \mathrm{skel}(E)) \\
\mathrm{skel}(FE, SE) &= (\mathrm{skel}(FE), \mathrm{skel}(SE)) \\
\mathrm{skel}(\{strid_1 \mapsto S_1, \ldots, strid_k \mapsto S_k\}) &= \{strid_1 \mapsto \mathrm{skel}(S_1), \ldots, \\
&\qquad\qquad strid_k \mapsto \mathrm{skel}(S_k)\} \\
\mathrm{skel}(\{funid_1 \mapsto \Phi_1, \ldots, funid_k \mapsto \Phi_k\}) &= \{funid_1 \mapsto \Phi_0, \ldots, funid_k \mapsto \Phi_0\}
\end{aligned}$$

where Φ_0 is a arbitrary closed functor signature.

Definition 3 (Well-formedness)
A semantic object $\mathcal{O}$ is *well-formed* if

1. all semantic objects occurring inside $\mathcal{O}$ are well-formed;
2. if $\mathcal{O}$ is signature $(N)S$ then $N \subseteq \text{names}(S)$, and also, whenever (m, E) occurs free in S and $m \notin N$, then $N \cap \text{names}(\text{skel}(E)) = \emptyset$;
3. if $\mathcal{O}$ is a functor signature $(N)(S, (N')S')$ then $(N)S$ is well-formed and also, whenever (m', E') occurs free in S' and $m' \notin N \cup N'$, then $N \cap \text{names}(\text{skel}(E')) = \emptyset$;

The reader might wonder whether the condition $N \cap \text{names}(\text{skel}(E)) = \emptyset$ in item (2) of the above definition is equivalent to the simpler $N \cap \text{names}(E) = \emptyset$. The chosen definition admits strictly more signature expressions than the simpler one. For an example, consider the declarations

```
structure A = struct functor f(X: sig end) = struct end end
signature Sig =
  sig
    local structure B: sig end
    in functor f(X: sig end is B' sharing B' = B) sig end
    end
  end is A' sharing A' = A
```

Here `Sig` denotes the signature

$$\left(\{m_2\}\right)\left(m_1, \left\{\mathrm{f} \mapsto \left(\emptyset\right)\left((m_2, \{\}), (\{m_3\})(m_3, \{\})\right)\right\}\right)$$

where m_1 is the name of `A`. This signature is well-formed according to the chosen definition, but not according to the simplified one. Note that consistency does not force the functor components of `A'` and `A` to have the same functor signatures. It appears that if local and overlapping specifications were omitted from the language, the simpler definition could be chosen without affecting the class of legal signature expressions. In the presence of local and overlapping specifications and the liberal notion of consistency of functor components, we need to admit the above signature as a well-formed signature, in order for the principality theorem to hold.

As a general principle, we wish to rule out elaborations of the form $A, B \vdash sigexp \Rightarrow \Sigma$ where Σ is unmatchable. Such signature expressions are useless. Indeed, they are obstructive to practical programming, when used as parameter signatures in functors. Thus a signature expression will be illegal if it specifies sharing between rigid

```
structure A = struct end;
structure B = struct end;
signature BADSIG1 =
sig
  sharing A = B
end;
```

(a)

```
structure A = struct end;
signature BADSIG2 =
sig
  structure A' :
    sig structure B: sig end end
  sharing A' = A
end;
```

(b)

Fig. 13. The two signatures above are illegal. In the case of (a), there is an attempt to identify to different rigid structures; in the case of (b), there is an attempt to specify a non-existent component of a rigid structure.

```
sig
  structure P1: sig structure Q: sig end end
  structure P2: sig end
  structure P3: sig structure Q: sig end end
  sharing P1=P2 sharing P2=P3
end
```

Fig. 14. Consistency and cover together ensure that sharing equations become transitive. Hence the above signature expression implicitly specifies sharing between `P1.Q` and `P3.Q` .

```
sig
  structure P: sig structure Q: sig end end
  sharing P.Q=P
end
```

Fig. 15. Cyclic structures cannot be declared and are therefore also banned in signatures.

structures that are manifestly different, see Fig. 13(a) for an example. Furthermore, a signature expression will be illegal if it postulates the existence of components that are manifestly nonexistent, see Fig. 13 (b) for an example.

More ambitiously, we wish to ensure that an elaboration $A, B \vdash sigexp \Rightarrow (N)S$ is possible only if the entire elaboration tree which has $A, B \vdash sigexp \Rightarrow (N)S$ as its conclusion does not involve such bad sharing specifications. We achieve this by arranging that every elaboration tree which proves $A, B \vdash sigexp \Rightarrow (N)S$ is of the form

$$\begin{array}{c} \vdots \\ A', B \vdash sigexp \Rightarrow S \\ | \\ A, B \vdash sigexp \Rightarrow (N)S \end{array}$$

where $N \cap \text{names}(A, B) = \emptyset$ and A' is a consistent assembly which covers A, S and all free structures above the node $A', B \vdash sigexp \Rightarrow S$. If *sigexp* has no local or overlapping sequential specifications, then simply taking $A' = (S, A)$ will do.

Example 3.1 Let *sigexp* be the signature expression in Fig. 14. Let A be the empty assembly and B the empty basis. To obtain $A', B \vdash sigexp \Rightarrow S$ with A' consistent, we must make sure not just that `P1`, `P2` and `P3` have the same name, but also that `P1.Q` and `P3.Q` have the same name, even though we have not explicity stated sharing between `P1` and `P3`. Consistency, as we have defined it, is not a transitive relation. But consistency combined with cover makes sharing equations transitive. □

As in Standard ML, we shall also ban cyclic specifications, like the one in Fig. 15, for such a specification cannot be matched by any real structure.

Definition 4 (Cycle-freedom)
A semantic object $\mathcal{O}$ is *cycle-free* if (after changing bound names to make all nameset prefixes in $\mathcal{O}$ disjoint) it contains no cycle of structure names; that is, there is no sequence $m_0, \cdots, m_{k-1}, m_k = m_0$, $(k > 0)$, of structure names such that, for each i $(0 \leq i < k)$ some structure with name m_i occurring in $\mathcal{O}$ has a proper substructure with name m_{i+1}.

The definitions of well-formedness, cycle-freedom and consistency easily extend to pairs of semantic objects. The conjunction of these three concepts is called admissibility:

Definition 5 (Admissibility)
A semantic object (typically an assembly) A is *admissible* if it is consistent, cycle-free and well-formed. We say that A_2 is an *admissible cover of* A_1, written $A_1 \sqsubseteq A_2$, if the pair (A_1, A_2) is admissible and A_2 covers A_1. We say that A_2 is a *conservative admissible cover* of A_1, written $A_1 \trianglelefteq A_2$, if the pair (A_1, A_2) is admissible and A_2 is a conservative cover of A_1. We say that A and A' are *equivalent*, written $A_1 \sim A_2$, if $A_1 \sqsubseteq A_2$ and $A_2 \sqsubseteq A_1$.

Both $\sqsubseteq$ and $\trianglelefteq$ are preorders. Note that $A \trianglelefteq A'$ implies $A \sqsubseteq A'$.

When A is an admissible assembly then the definition of Graph(A) makes sense: the well-formedness of A ensures that whenever m is a node in the graph (i.e. $m \in \text{names}(A)$) and $(m, E) = (m, (FE, SE))$ occurs free in A and $(m', E') = SE(strid)$, for some *strid*, then m' is free in A and therefore a node in the graph. Moreover, Graph(A) is a directed acyclic graph satisfying that whenever

$$m \xrightarrow{strid} m' \qquad m \xrightarrow{strid} m''$$

are both in the graph then $m' = m''$.

3.4 Realization

The approach to signature elaboration which we outlined above is not particularly operational, as one has to 'guess' a good conservative cover A' of A when *sigexp* is to be elaborated in A. However, as we shall see later, an algorithm can gradually build up an assembly during a syntax-directed traversal of the signature expression. The algorihm must be able to add components to flexible (i.e. specified) structures, although it must not add components to rigid (e.g. declared) structures.

When the algorithm meets a sharing equation, it invokes a unification algorithm on the current assembly and tries to identify the names of the structures that are specified to share. This may recursively involve identification of names of common substructures. The unification either fails (by which is meant that it aborts with a special message **fail**) or it produces a substitution from structure names to structure names.

We shall express the distinction between rigid and flexible structures by keeping the names of all the structures that are considered rigid in a special place in the basis: a structure name m is *rigid in B*, if $m \in N$ of B.

The substitutions produced by structure unification are referred to as *realizations* (Harper *et al.*, 1987; Tofte, 1988; Milner *et al.*, 1990):

Definition 6 (Realization)
Let φ be a map φ : StrName $\rightarrow$ StrName. The *support* of φ, written Supp(φ), is the set of names m such that $\varphi(m) \neq m$. The map φ is a realization if Supp(φ) is finite. The *yield* of φ, written Yield(φ), is the set $\{\varphi(m) \mid m \in \text{Supp}(\varphi)\}$.

Realizations φ are extended to apply to all semantic objects; their effect is to replace each free name n by $\varphi(n)$. In applying φ to an object with bound names, such as a signature $(N)S$, first bound names must be changed to avoid name capture. Application is extended to name sets N as follows: $\varphi(N) = \{\varphi(n) \mid n \in N\}$. We often omit parentheses from applications: φA means $\varphi(A)$. A realization φ is *fixed on* N if $\varphi(n) = n$, for all $n \in N$.

In polymorphic type disciplines one usually has a substitution lemma along the lines: for all substitutions S, if $A \vdash e : \tau$ then $S(A) \vdash e : S(\tau)$. One reason why such a lemma is important is that it is precisely by applying a substitution to a typing statement $A \vdash e : \tau$ that the textual context of e can influence the typing of e. Let us try to use the same idea on elaboration statements $A, B \vdash sigexp \Rightarrow S$. Here the idea would be that applying a realization φ to this statement should correspond to letting sharing equations in the textual context of *sigexp* influence the typing of *sigexp*. Just applying a realization to A and B can certainly identify names, but we also need to be able to 'add' components to flexible structures in A; the latter operation is less trivial to model by substitution, as is known from work on polymorphic record typing (Wand, 1989; Rémy, 1989). It is possible to use a record type discipline to express the widening of structures (Aponte, 1992, 1993). We have chosen a different approach, namely to consider φ to be a relation between semantic objects, expressed in terms of (very elementary) category theory.

Definition 7 (The category K)
K is the category defined as follows. An object O of K is a pair $(A, B) \in \text{Asmb} \times \text{Basis}$ satisfying $A \sqsupseteq B$. The set of objects of K is denoted Obj. For all objects $O_1 = (A_1, B_1) = (A_1, (N_1, G_1, E_1))$ and $O_2 = (A_2, B_2)$, and for every realization φ, there is a morphism $O_1 \xrightarrow{\varphi} O_2$ if φ is fixed on N_1, $\varphi(B_1) = B_2$, $\varphi(A_1) \sqsubseteq A_2$ and A_1 covers A_2 on N_1.

The condition 'A_1 covers A_2 on N_1' prevents realization from adding components to any structure, whose name is rigid in B_1 — a sensible condition since a rigid structure cannot be extended with more structures, once generated. Notice that $O_1 \xrightarrow{\varphi} O_2$ implies $\varphi(B_1) = \varphi(N_1, G_1, E_1) = (N_1, \varphi G_1, \varphi E_1) = B_2$. Thus, even though $O_1 \xrightarrow{\varphi} O_2$ can 'widen' structures in A_1, it does not affect the shape of structures in B_1 — once structures get into the basis, they do not change shape.

It is convenient not to require $\text{Supp}(\varphi) \subseteq \text{names}(O)$ in the above definition. Morphisms $O_1 \xrightarrow{\varphi} O_2$ and $O_1 \xrightarrow{\varphi'} O_2$ between O_1 and O_2 are equal if $\varphi(n) = \varphi'(n)$, for all $n \in \text{names}\, O_1$. Notice that this is weaker than demanding $\varphi = \varphi'$. Composition in K is the natural extension of composition of realizations. Thus, that a diagram of the form

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

commutes does not imply $\varphi = \psi \circ \varphi^\bullet$, but it does imply that the restrictions of these two maps to the set names(O) are equal.

3.5 Principal signatures

The inference rules define what elaborations are possible, without saying *how* one decides whether a given signature expression elaborates. The advantage of using such 'liberal' inference rules is that one can give rules for sharing without having to spell out unification or rules for applying realizations. This is similar to the situation in Milner's polymorphic type discipline, where liberal inference rules are used without reference to any particular unification procedure.

In both disciplines the question arises, whether one can elaborate phrases to a 'best' (or 'principal') result. In the Damas–Milner type discipline (Damas and Milner, 1982) it is the case that every expression elaborates to a principal type scheme, if it elaborates at all. Similarly, for first-order ML modules, a signature elaborates to a *principal signature*, if it elaborates at all (Tofte, 1988; Milner and Tofte, 1991).

The existence of principal signatures is more than just evidence of a certain technical coherence in the semantics; it is also essential for the practical use of the modules system. Assume that *sigexp* is the parameter signature of some functor F and that *sigexp* elaborates to a principal signature $\Sigma = (N)S$. From Σ there is a simple way of achieving a structure which has precisely the sharing and the components that every structure which matches *sigexp* must have — in other words, from Σ one can get a 'template' structure which can be used as a prototype of all structures that match *sigexp*. Indeed, in the first order language, S is precisely such a structure (provided $\Sigma = (N)S$ is principal for *sigexp* and the name set N is suitably 'new'). It is therefore possible to elaborate the body of F under the assumption that the formal parameter of F is bound to S. If there were no principal signature, but perhaps five good candidates for S, which one should be used inside the body of F? The ability to elaborate functor declarations (without knowing the identity of the structures it will later be applied to) is crucial to the practical use of the modules system (see the Introduction). Thus it is worth the effort to design the language in such a way that one can prove that principal signatures exist.

We shall now define the notion of principal signature formally. First, let us say that a structure S' is an *instance* of a signature $\Sigma = (N)S$, written $\Sigma \geq S'$, if there exists a realization φ such that $\mathrm{Supp}(\varphi) \subseteq N$ and $\varphi(S) = S'$.

Definition 8 (Principal signature)
We say that a signature Σ is *principal for sigexp in* $O = (A, B)$ if $O \in \mathrm{Obj}$ and, writing Σ in the form $(N)S$ where $N \cap \mathrm{names}(A) = \emptyset$,

1. There exists an A' such that $A \trianglelefteq A'$ and $A', B \vdash \mathit{sigexp} \Rightarrow S$
2. For all O', φ and S', if $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{sigexp} \Rightarrow S'$ then $\varphi(\Sigma) \geq S'$

Item 1 of this definition should be fairly natural on the background of the discussion in the previous sections. To understand item 2, consider the special case where φ is the identity map Id. Then the condition is that if $(A, B) \in \mathrm{Obj}$ and $(A', B) \in \mathrm{Obj}$ and $A \sqsubseteq A'$ and A covers A' on the N-set of B and $A', B \vdash \mathit{sigexp} \Rightarrow S'$ then $\Sigma \geq S'$. In other words, even if the context should later widen flexible structures in A, Σ would still be general enough that all possible results of the elaboration in the widened assembly could be obtained by instantiation of Σ. This may seem surprising at first, since *sigexp* can specify sharing with some of the flexible structures in the basis. However, recall that structures are only being widened in the assembly, not in the basis; the free structures of a principal signature all stem from the basis, so they will not have been widened.

Now item (2) merely says that principality is preserved not just under widening of flexible structures in the assembly, but also under full realization, as defined in the category K.

One might ask whether the condition $A \trianglelefteq A'$ in item 1 could be $A \sqsubseteq A'$ instead. Here the answer is that the latter would be too weak. We want to ensure that A covers not just Σ but also structures that occur (free) in the proof of $A, B \vdash sigexp \Rightarrow \Sigma$ and share with structures in A. (Recall that in general, when `local` and overlapping sequential specifications are allowed, not all such structures need be visible in Σ.)

We can now state the principality theorem:

Theorem 3.1 (Principal signatures)
Let B be a basis, A an assembly, let $A \sqsupseteq B$, $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S$, for some S. Then there exists a principal signature for *sigexp* in A, B.

3.6 Functor signatures

A functor specification

$$\texttt{functor}\ funid : (strid : sigexp_1)\, sigexp_2 \qquad (3)$$

specifies a functor which it must be possible to apply to any structure which matches $sigexp_1$; moreover, the functor must satisfy that the result of the application matches $sigexp_2$. It is possible to specify sharing between argument and result. It is also possible to specify sharing with structures specified or declared outside the functor specification. If legal, the specification (3) elaborates to a functor environment of the form $\{funid \mapsto \Phi\}$ where Φ is a functor signature $(N_1)(S_1,(N_2)S_2)$. Here $(N_1)S_1$ is the principal signature for $sigexp_1$; no other signature for $sigexp_1$ is of interest, for $(N_1)S_1$ accurately captures what every structure must satisfy in order to match $sigexp_1$. Moreover, $(N_2)S_2$ is the principal signature for $sigexp_2$ in a basis in which *strid* has been bound to S_1. Since there may be sharing between argument and result, the scope of (N_1) is both S_1 and $(N_2)S_2$. Because $(N_2)S_2$ is required to be principal, only those names that have been specified to share with the formal parameter *strid* (or with external structures) will be free in $(N_2)S_2$.

Example 3.2 Consider the (legal) specification of F in Fig. 16(b). Let $S = (m, \{\texttt{Q} \mapsto (m', \{\})\})$, let $B = (\emptyset, \{\}, \{\texttt{P} \mapsto S\})$ and let A be just S. The functor signature expression for F elaborates to the following functor signature in A, B: $(\emptyset)((m,\{\}),(\emptyset)S)$. If the sharing qualifications were dropped, it would elaborate to the functor signature $(\{m_1\})((m_1,\{\}),(\{m_2,m_3\})(m_2,\{\texttt{Q} \mapsto (m_3,\{\})\}))$ □

It is important that *funid* can be applied to any structure which matches $sigexp_1$. Therefore, we cannot allow $sigexp_2$ to widen a structure whose name is in N_1 — see Fig. 16(a) for an example. On the other hand, $sigexp_2$ can contribute components

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

```
sig
  functor F:
     (X: sig end)
     sig structure Y:sig end
     end is X' sharing X'=X
end
```

(a)

```
sig
  structure P: sig end
  functor F:
     (X: sig end is X' sharing X' = P)
     sig
       structure Q: sig end
     end is R sharing R=X
end
```

(b)

Fig. 16. Sharing between argument and result in a functor specification cannot contribute components to a structure whose name is quantified in Σ_1, the principal signature for the argument signature expression. Hence (a) is illegal. However, sharing can expand the assembly with new components of structures that are free in Σ_1, so (b) is legal.

to flexible structures in the assembly in which it is elaborated — see Fig. 16(b) for an example. This is manifest in the inference rule:

$$\frac{A, B \vdash \mathit{sigexp}_1 \Rightarrow (N_1)S_1 \quad N_1 \cap \mathrm{names}\, A = \emptyset \qquad (S_1, A), B + N_1 + \{\mathit{strid} \mapsto S_1\} \vdash \mathit{sigexp}_2 \Rightarrow \Sigma_2}{A, B \vdash (\mathit{strid}\!:\!\mathit{sigexp}_1)\, \mathit{sigexp}_2 \Rightarrow (N_1)(S_1, \Sigma_2)}$$

Note that we extend the assembly with S_1. Since (S_1, A) has to cover the elaboration of *sigexp*$_2$, any N_1-bound structure postuated to exist by *sigexp*$_2$ must really be in S_1. In other words, the argument signature specification must specify all the N_1-bound structures that *sigexp*$_2$ refers to. Also, since we add the set N_1 to the rigid names of B, realization on (A, B) cannot add components of N_1-bound structures to the assembly.

3.7 Inference rules

The inference rules appear below. All the conclusions of the rules are of the form

$$A, B \rhd \mathit{phrase} \Rightarrow P$$

Here *phrase* is a specification, signature expression, functor signature expression or a sharing equation.

In the premises of the rules, '$A, B \vdash \mathit{phrase} \Rightarrow P$' abbreviates '$A \sqsupseteq (B, P)$ and $A, B \rhd \mathit{phrase} \Rightarrow P$'; for example, rule 4 in its expanded form is

$$\frac{A \sqsupseteq (B, E) \quad A, B \rhd \mathit{spec} \Rightarrow E}{A, B \rhd \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)}$$

We say that *phrase elaborates to P in* (A, B), written $A, B \vdash \mathit{phrase} \Rightarrow P$, if $A \sqsupseteq (B, P)$ and there is an inference tree which satisfies all the side-conditions on the rules and has $A, B \rhd \mathit{phrase} \Rightarrow P$ as its conclusion.

There is one rule for each production in Fig. 1, plus rule 7, which concerns principal signatures. A sample elaboration is shown after the inference rules.

Signature expressions $\boxed{A, B \rhd sigexp \Rightarrow S}$

$$\frac{A, B \vdash spec \Rightarrow E}{A, B \rhd \texttt{sig}\ spec\ \texttt{end} \Rightarrow (m, E)} \tag{4}$$

$$\frac{B(sigid) \geq S}{A, B \rhd sigid \Rightarrow S} \tag{5}$$

$$\frac{A, B \vdash sigexp \Rightarrow S \quad A, B + \{strid \mapsto S\} \vdash shareq \Rightarrow \{\}}{A, B \rhd sigexp\ \texttt{is}\ strid\ \texttt{sharing}\ shareq \Rightarrow S} \tag{6}$$

$\boxed{A, B \rhd sigexp \Rightarrow \Sigma}$

$$\frac{\Sigma \text{ principal for } sigexp \text{ in } A, B}{A, B \rhd sigexp \Rightarrow \Sigma} \tag{7}$$

Functor signature expressions $\boxed{A, B \rhd funsigexp \Rightarrow \Phi}$

$$\frac{\begin{array}{c} A, B \vdash sigexp_1 \Rightarrow (N)S \quad N \cap \text{names}\, A = \emptyset \\ (S, A), B + N + \{strid \mapsto S\} \vdash sigexp_2 \Rightarrow \Sigma \end{array}}{A, B \rhd (strid{:}sigexp_1)sigexp_2 \Rightarrow (N)(S, \Sigma)} \tag{8}$$

Atomic specifications $\boxed{A, B \rhd atspec \Rightarrow E}$

$$\frac{A, B \vdash sigexp \Rightarrow S}{A, B \rhd \texttt{structure}\ strid{:}sigexp \Rightarrow \{strid \mapsto S\}} \tag{9}$$

$$\frac{A, B \vdash funsigexp \Rightarrow \Phi}{A, B \rhd \texttt{functor}\ funid{:}funsigexp \Rightarrow \{funid \mapsto \Phi\}} \tag{10}$$

$$\frac{A, B \vdash shareq \Rightarrow \{\}}{A, B \rhd \texttt{sharing}\ shareq \Rightarrow \{\}} \tag{11}$$

$$\frac{A, B \vdash spec_1 \Rightarrow E_1 \qquad A, B + E_1 \vdash spec_2 \Rightarrow E_2}{A, B \rhd \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow E_2} \tag{12}$$

Specifications $\boxed{A, B \rhd spec \Rightarrow E}$

$$\frac{}{A, B \rhd \qquad \Rightarrow \{\}} \tag{13}$$

$$\frac{A, B \vdash atspec_1 \Rightarrow E_1 \qquad A, B + E_1 \vdash spec_2 \Rightarrow E_2}{A, B \rhd atspec_1\ \langle ; \rangle\ spec_2 \Rightarrow E_1 + E_2} \tag{14}$$

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

```
structure P =
struct
 structure Q = struct end
end;

signature SIG =
sig(3)
 structure P' : sig end
 functor F: (X:
  sig(4)
   structure P'':
    sig
     structure Q: sig end
    end
   sharing P'' = P'
  end(4)): sig end
 sharing P' = P
end(3);
```

Fig. 17. Embedded functor signature.

Sharing equations $\boxed{A, B \rhd shareq \Rightarrow \{\}}$

$$\frac{m \text{ of } B(longstrid_1) = m \text{ of } B(longstrid_2)}{A, B \rhd longstrid_1 \; = \; longstrid_2 \Rightarrow \{\}} \tag{15}$$

Rules 7 and 8 have already been explained. In rule 4 we can freely choose m so as to satisfy the side-condition on rule 15. Similarly, rule 5 allows us to take any instance of the signature $B(sigid)$.

In rule 7, the side-condition that Σ be principal for *sigexp* in A, B implies that there exists an A' such that $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S$ and $\Sigma = (N)S$, for some N with $N \cap \text{names}(A) = \emptyset$. In doing inductive proofs on the depth of inference, we include the depth of the proof of $A', B \vdash sigexp \Rightarrow S$ when we count the depth of the proof of $A, B \vdash sigexp \Rightarrow \Sigma$. This makes sense because the depth of a proof of $A', B \vdash sigexp \Rightarrow S$ can be determined from the syntactic structure of *sigexp* alone.

As an example of the use of the rules, the elaboration of the program in Fig. 17 is summarised in Fig. 18. It illustrates most features of the inference system. The specifications of `P'` and `F` are referred to as $spec_{\texttt{P'}}$ and $spec_{\texttt{F}}$, respectively. The signature expressions $\texttt{sig}^{(i)} \ldots \texttt{end}^{(i)}$ are abbreviated $sigexp_i$ $(i = 3, 4)$. We assume that the assembly and the basis are empty at the outset. After the elaboration of the structure declaration we have $B = \{\texttt{P} \mapsto S_{\texttt{P}}\}$ and $A = [S_{\texttt{P}}]$, where $S_{\texttt{P}} = (\texttt{m1}, \{\texttt{Q} \mapsto (\texttt{m2}, \{\})\})$. The elaboration then proceeds as outlined in Fig. 18.

Lemma 3.1

Let $A_1 \sim A_2$. Then Σ is principal for *sigexp* in (A_1, B) if and only if Σ is principal for *sigexp* in (A_2, B). Moreover, $A_1, B \vdash phrase \Rightarrow P$ if and only if $A_2, B \vdash phrase \Rightarrow P$.

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

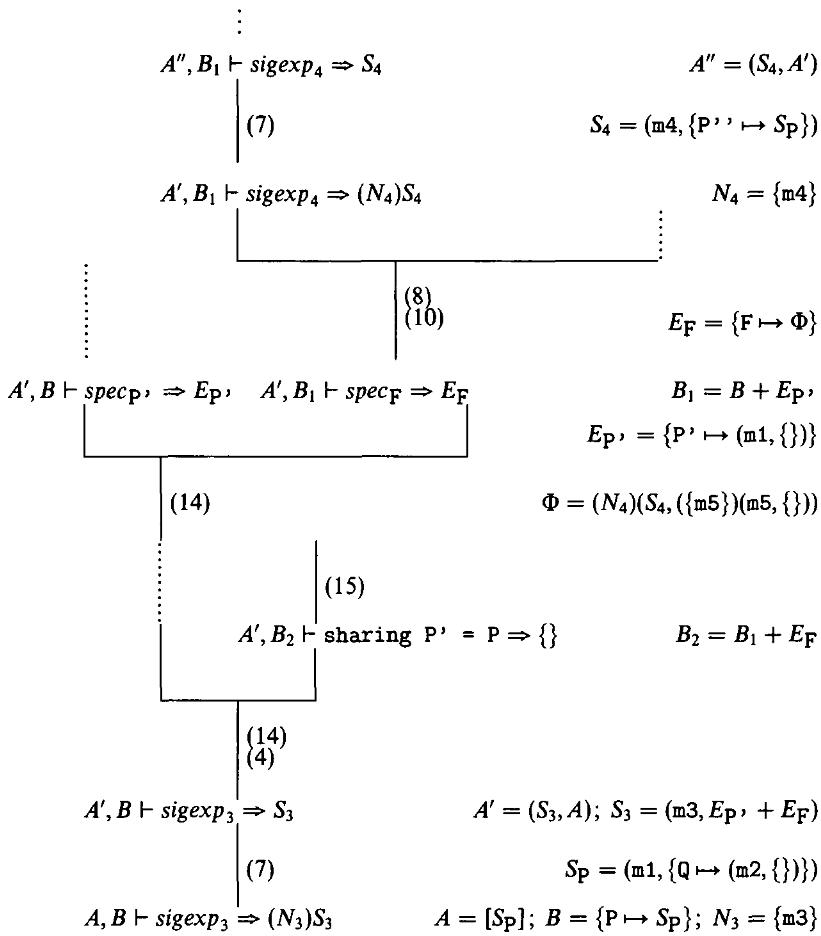


Fig. 18. An elaboration tree.

In other words, it is the graph of an assembly that matters to elaboration, rather than the assembly itself.

The side-conditions concerning admissibility and cover may have left the reader wondering whether an implementation has to enforce these constraints every time it tries to make an inference step. This is not the case. Indeed, assuming that the input (A, B) to the algorithm satisfies $A \sqsupseteq B$, then any attempt to violate the side-conditions can be detected by the unification algorithm which deals with the sharing equations.

We have now completed our presentation of the semantics. The rest of the paper is organised as follows. In section 4 we discuss structure unification. In section 5.1 we prove that elaboration is preserved under realization and in section 6 we present an algorithm for inferring principal signatures and prove it correct.

4 Unification

Finding a principal signature involves solving sharing equations. Consider the problem of deciding, for given assembly A and basis B, whether the sharing equation

$$longstrid_1 = longstrid_2 \tag{16}$$

can be satisfied. Assume $B(longstrid_1) = (m_1, E_1)$ and $B(longstrid_2) = (m_2, E_2)$. We wish to obtain that m_1 and m_2 are equal (cf. the side-condition on rule 15). Thus we seek a realization φ such that $\varphi(m_1) = \varphi(m_2)$. Since the sharing constraint (16) implicitly specifies sharing between all common substructures of $longstrid_1$ and $longstrid_2$ (whether visible or not), φ may have to make further name identifications. To be precise, we want $\varphi(A)$ to be admissible; the well-formedness of $\varphi(A)$ follows from the well-formedness of A so it is the consistency and cycle-freedom of $\varphi(A)$ that is important here.

But φ has to satisfy more. First, φ must be fixed on N of B, i.e. we must have $\varphi(m) = m$, for all $m \in (N \text{ of } B)$. Also, applying a realization must not have the effect of widening rigid structures, i.e. we must have that A covers $\varphi(A)$ on N of B. We collect these properties in the following definition:

Definition 9 (Unifier)
Let A be an admissible assembly, let m_1 and m_2 be names that occcur free in A and let N be a name set. A realization φ is a *unifier for m_1 and m_2 in A under N* if $\varphi(m_1) = \varphi(m_2)$, $\varphi(A)$ is admissible, φ is fixed on N and A covers $\varphi(A)$ on N.

As in the case of ordinary first-order term unification, there is a notion of most general unifier:

Definition 10 (Most general unifier)
Let φ^* be a unifier for m_1 and m_2 in A under N. Then φ^* is said to be *most general* if whenever φ is a unifier for m_1 and m_2 in A under N there exists a realization φ' which is fixed on N and satisfies $\varphi'(\varphi^*(A)) = \varphi(A)$.

Also, there exists an algorithm *Unify* which satisfies the following property:

Theorem 4.1 (Unification)
Let A be an admissible assembly, N be a name set, and let m_1 and m_2 be names occurring free in A. If there exists some unifier for m_1 and m_2 in A under N then $Unify(A, N, (m_1, m_2))$ returns a most general unifier for m_1 and m_2 in A under N. Otherwise, $Unify(A, N, (m_2, m_2))$ fails.

We shall not spell out the details of the algorithm here, nor shall we prove the above theorem, for there are several similar algorithms and proofs in the literature, e.g. algorithms by Ait-Kaci (1986), Rémy (1989) and Aponte (1992). The following outline of an algorithm is based on the Commentary of Standard ML (Milner and Tofte, 1991), which we refer to as 'the Commentary' in what follows. *Unify* first builds the smallest equivalence relation $\equiv$ on names(A) satisfying that (a) $m_1 \equiv m_2$ and (b) for all m, n, m', n' and $strid$ if $m \equiv n$ and

$$m \xrightarrow{strid} m' \qquad n \xrightarrow{strid} n'$$

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

are both in Graph(A) then $m' \equiv n'$. From $\equiv$ it is easy to see whether a unifier exists; if it does, a most general unifier with kernel $\equiv$ is returned. In HML, failure can only happen because of an attempt to identify two different rigid names (i.e. two names that are both in N) or an attempt to add a component to a rigid structure.

Notice that unification does not have to make functor components equal. This is because consistency (Definition 1) does not require functor components to be equal.

5 The Realization Theorem

In this section we prove that elaboration is preserved under realization. In order to prove this fact, we need the following lemma, which is taken from the Commentary (Milner and Tofte, 1991).

Lemma 5.1

For any signature Σ, structure S and realization φ, if $\Sigma \geq S$ then $\varphi(\Sigma) \geq \varphi(S)$.

Proof

Write Σ as $(N)S'$, assuming w.l.o.g. that $(\mathrm{Supp}\,\varphi \cup \mathrm{Yield}\,\varphi) \cap N = \emptyset$. Then $\varphi\Sigma = \varphi((N)S') = (N)\varphi S'$. So it suffices to find a realization ψ such that $\mathrm{Supp}\,\psi \subseteq N$ and $\psi(\varphi(S')) = \varphi(S)$.

Now since $\Sigma \geq S$ there exists ψ' such that $\mathrm{Supp}(\psi') \subseteq N$ and $\psi'(S') = S$. Define ψ to be the restriction of $\varphi \circ \psi'$ to N, i.e. $\psi n = \varphi(\psi'(n))$ if $n \in N$ and $\psi(n) = n$ if $n \notin N$.

To prove $\psi(\varphi(S')) = \varphi(S)$, it is enough to show that $\psi(\varphi(n)) = \varphi(\psi'(n))$ for every $n \in \mathrm{names}(S')$; this is now straightforward, considering the two cases $n \in N$ and $n \notin N$ separately. $\square$

We also need the following two lemmas, which state that cover and admissible cover are preserved under realization:

Lemma 5.2

If A_2 covers A_1 then $\varphi(A_2)$ covers $\varphi(A_1)$.

Proof

We have to prove that $\varphi(A_2)$ covers $\varphi(A_1)$ on names($\varphi(A_1)$). Since names$(A_1) \subseteq$ names(A_2) we have names$(\varphi(A_1)) \subseteq$ names$(\varphi(A_2))$ as required. Let (m, E_1) be a structure occurring free in $\varphi(A_1)$, and let *id* be a structure- or functor identifier in the domain of E_1. Then $(m, E_1) = \varphi(m', E_1')$, for some (m', E_1') occurring free in A_1. Since $id \in \mathrm{Dom}(E_1')$ and A_2 covers A_1, there exists a E_2' such that (m', E_2') occurs free in A_2 and $id \in \mathrm{Dom}(E_2')$. Thus $\varphi(m', E_2') = (m, \varphi(E_2'))$ occurs free in $\varphi(A_2)$ with $id \in \mathrm{Dom}(\varphi(E_2'))$, showing that $\varphi(A_2)$ covers $\varphi(A_1)$. $\square$

Lemma 5.3

If $A_1 \sqsubseteq A_2$ and $\varphi(A_2)$ is admissible, then $\varphi(A_1) \sqsubseteq \varphi(A_2)$.

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

Proof

We have that $\varphi(A_2)$ covers $\varphi(A_1)$, by Lemma 5.2. It remains to prove that the pair $(\varphi(A_2), \varphi(A_1))$ is admissible. For all assemblies A, if A is well-formed, then so is $\varphi(A)$. In particular, $(\varphi(A_2), \varphi(A_1))$ is well-formed. Also, $(\varphi(A_2), \varphi(A_1))$ is consistent, because (A_2, A_1) is consistent, A_2 covers A_1 and $\varphi(A_2)$ is consistent. Finally, $(\varphi(A_2), \varphi(A_1))$ is cycle-free, because $\varphi(A_2)$ is cycle-free and $\varphi(A_2)$ covers $\varphi(A_1)$. □

Theorem 5.1

Let *phrase* be a signature expression, a specification, a sharing equation or a functor signature expression. If $O \vdash \mathit{phrase} \Rightarrow P$ and $O \xrightarrow{\varphi} O'$ then $O' \vdash \mathit{phrase} \Rightarrow \varphi(P)$.

Proof

We use induction on the depth of inference. There is one case for each inference rule. In all the cases the argument that $O' \rhd \mathit{phrase} \Rightarrow \varphi(P)$ implies $O' \vdash \mathit{phrase} \Rightarrow \varphi(P)$ is the same: since $O \vdash \mathit{phrase} \Rightarrow P$ we have $P \sqsubseteq (A \text{ of } O)$; thus by Lemma 5.3, $\varphi(P) \sqsubseteq \varphi(A \text{ of } O) \sqsubseteq A \text{ of } O'$, since $O \xrightarrow{\varphi} O'$. Thus $\varphi(P) \sqsubseteq A \text{ of } O'$, showing $O' \vdash \mathit{phrase} \Rightarrow \varphi(P)$. In each case it will therefore suffice to prove $O' \rhd \mathit{phrase} \Rightarrow \varphi(P)$. The cases for rules (4), (6) and (9)–(15) are all straightforward arguments. We show only the first of these, as an example.

$\boxed{\text{Rule 4, } \mathit{sigexp} \equiv \texttt{sig } \mathit{spec} \texttt{ end}}$

Let $(A, B) = O$ and $(A', B') = O'$. Assume $O \xrightarrow{\varphi} O'$ and $O \vdash \texttt{sig } \mathit{spec} \texttt{ end} \Rightarrow S$. Then $S = (m, E)$ and $O \vdash \mathit{spec} \Rightarrow E$, for some m and E. By induction we have $O' \vdash \mathit{spec} \Rightarrow \varphi E$. Thus $O' \rhd \texttt{sig } \mathit{spec} \texttt{ end} \Rightarrow (\varphi m, \varphi E)$, by rule 4.

$\boxed{\text{Rule 5, } \mathit{sigexp} \equiv \mathit{sigid}}$

Let $(A, B) = O$ and $(A', B') = O'$. Assume $O \xrightarrow{\varphi} O'$ and $O \vdash \mathit{sigid} \Rightarrow S$. By rule 5 we have $B(\mathit{sigid}) \geq S$. Thus $(\varphi B)(\mathit{sigid}) \geq \varphi S$, i.e. $B'(\mathit{sigid}) \geq \varphi S$, by Lemma 5.1.

$\boxed{\text{Rule 7, principal signatures}}$

Let $(A, B) = O$ and $(A', B') = O'$. By rule 7, P is a principal signature for *sigexp* in O. Thus P can be written in the form $(N)S$, where $N \cap \text{names}\, A = \emptyset$, and there exists an A_1 such that $A \trianglelefteq A_1$ and

$$A_1, B \vdash \mathit{sigexp} \Rightarrow S \tag{17}$$

A_1 contains all the names that occur free in S, including the ones in N. We cannot simply apply φ to (17) and expect to get an elaboration, for we know nothing about the behaviour of φ outside names(A). We therefore apply induction on a realization φ_1 which coincides with φ on names(A) and maps names in names(A_1) $\setminus$ names(A) to distinct fresh names. Formally, let $N_1 = \text{names}(A_1) \setminus \text{names}(A)$, let N_1' be a set of names satisfying that $N_1' \cap \text{names}(A') = \emptyset$ and that there are equally many names in N_1 and in N_1'. Let φ_1 be a realization satisfying that $\varphi_1 \downarrow (\text{names}(A)) = \varphi \downarrow (\text{names}(A))$ and that $\varphi_1 \downarrow N_1$ is an injective map from N_1 to N_1'. Let $A_1' = (A', \varphi_1(A_1))$. Since $A \trianglelefteq A_1$ and $N_1' \cap \text{names}(A') = \emptyset$ we have that A_1' is admissible. (Note that $A \sqsubseteq A_1$ would not have sufficed here.) Thus $(A_1', B') \in \text{Obj}$ and since $\varphi_1 B = \varphi B = B'$, we have $(A_1, B) \xrightarrow{\varphi_1} (A_1', B')$. By induction on (17) we therefore have

$$A_1', B' \vdash \mathit{sigexp} \Rightarrow \varphi_1 S$$

Let $N' = \varphi_1 N$. Note that $N \subseteq N_1$, as $N \cap \text{names}(A) = \emptyset$ and $A_1 \sqsupseteq S$. Also, $\text{names}((N)S) \subseteq \text{names}(A)$, as $O \vdash \textit{sigexp} \Rightarrow (N)S$. Thus $\varphi((N)S) = (N')(\varphi_1 S)$. Is this signature principal for *sigexp* in O'? Certainly $A' \trianglelefteq A'_1$ and $A'_1, B' \vdash \textit{sigexp} \Rightarrow \varphi_1 S$, as required. Also, $N' \cap \text{names}(A') = \emptyset$, as required. Finally, let O'', S' and ψ be such that $O' \xrightarrow{\psi} O''$ and $O'' \vdash \textit{sigexp} \Rightarrow S'$. Then $O \xrightarrow{\psi\circ\varphi} O''$. Since $(N)S$ is principal for *sigexp* in O we have that $(\psi \circ \varphi)((N)S) \geq S'$, i.e. $\psi((N')(\varphi_1 S)) \geq S'$, as required. Thus $(N')(\varphi_1 S)$ is principal for *sigexp* in O'. Thus we can apply rule 7 and get $O' \triangleright \textit{sigexp} \Rightarrow (N')(\varphi_1 S)$, i.e. $O' \triangleright \textit{sigexp} \Rightarrow \varphi((N)S)$.

Rule 8, $\textit{funsigexp} \equiv (\textit{strid} : \textit{sigexp}_1) : \textit{sigexp}_2$

Assume $O \xrightarrow{\varphi} O'$ and $O \vdash \textit{funsigexp} \Rightarrow \Phi$. Let $(A, B) = O$ and $(A', B') = O'$. Then *funsigexp* is of the form $(\textit{strid} : \textit{sigexp}_1) : \textit{sigexp}_2$ and Φ can be written $(N_1)(S_1, \Sigma_2)$, where $N_1 \cap \text{names}(O) = \emptyset$ and

$$A, B \vdash \textit{sigexp}_1 \Rightarrow (N_1)S_1 \tag{18}$$

$$(S_1, A), B + N_1 + \{\textit{strid} \mapsto S_1\} \vdash \textit{sigexp}_2 \Rightarrow \Sigma_2 \tag{19}$$

Without loss of generality we can assume that $N_1 \cap \text{names}(O, O') = \emptyset$ and that $\varphi n = n$, for all $n \in N_1$. By induction on (18) we have $A', B' \vdash \textit{sigexp}_1 \Rightarrow \varphi((N_1)S_1)$, i.e.

$$A', B' \vdash \textit{sigexp}_1 \Rightarrow (N_1)(\varphi S_1) \tag{20}$$

By the definition of $\vdash$ we therefore have that $A' \sqsupseteq (N_1)(\varphi S_1)$. But then, since $N_1 \cap \text{names}(A') = \emptyset$, we have that $(A', \varphi(S_1))$ is admissible. Also, $N_1 \subseteq \text{names}(\varphi(S_1))$ since the signature $(N_1)(\varphi S_1)$ is well-formed. Thus $((\varphi S_1, A'), B' + N_1 + \{\textit{strid} \mapsto \varphi S_1\}) \in \text{Obj}$. But then we have $(S_1, A), B + N_1 + \{\textit{strid} \mapsto S_1\} \xrightarrow{\varphi} (\varphi S_1, A'), B' + N_1 + \{\textit{strid} \mapsto \varphi S_1\}$, so by induction using (19) we have

$$(\varphi S_1, A'), B' + N_1 + \{\textit{strid} \mapsto \varphi S_1\} \vdash \textit{sigexp}_2 \Rightarrow \varphi \Sigma_2 \tag{21}$$

From (20) and (21) we get $A', B' \triangleright \textit{funsigexp} \Rightarrow (N_1)(\varphi S_1, \varphi \Sigma_2)$. But N_1 is chosen disjoint from $\text{names}(O, O')$ and $A \sqsupseteq (N_1)S_1$ and $(S_1, A) \sqsupseteq \Sigma_2$ (by (18) and (19), respectively), so $\varphi\Phi = (N_1)(\varphi S_1, \varphi \Sigma_2)$. $\square$

Notice that an algorithm which applies realization to a proof of $O \vdash \textit{phrase} \Rightarrow P$ does not have to make any checks for whether side-conditions concerning admissibility and cover are respected. As we saw in the beginning of the above proof, the side-conditions follow automatically from $O \xrightarrow{\varphi} O'$.

6 The inference algorithm

In this section we shall prove that if a signature expression can be elaborated at all, then it can be elaborated to a principal signature. The proof is constructive, in that we present an algorithm W and then prove that the algorithm really does find principal signatures. In section 6.1 we present the algorithm. In section 6.2 we restate the principality theorem (Theorem 3.1) in a stonger form suitable for inductive proof. We refer to this stronger version as the *main theorem*. In section 6.3

$W_{sigexp}(O \text{ as } (A, B), sigexp) : \mathrm{Obj} \times \mathrm{Rea} \times \mathrm{Str} =$
  case $sigexp$ of
    `sig` $spec$ `end` => (* rule 4 *)
      let $(O_1^\bullet \text{ as } (A_1^\bullet, B_1^\bullet), \varphi_1^\bullet, E_1^\bullet) = W_{spec}(O, spec)$
        $S_1^\bullet = (m^\bullet, E_1^\bullet)$, where $m^\bullet$ is new
        $O^\bullet = ((S_1^\bullet, A_1^\bullet), B_1^\bullet)$
      in $(O^\bullet, \varphi_1^\bullet, S_1^\bullet)$
  | $sigid$ => (* rule 5 *)
      if $sigid \notin \mathrm{Dom}(B)$ then **fail**
      else let $(N^\bullet)S^\bullet = B(sigid)$, where all names in $N^\bullet$ are new
        $O^\bullet = ((S^\bullet, A), B)$
        in $(O^\bullet, \mathrm{Id}, S^\bullet)$
  | $sigexp_1$ `is` $strid$ `sharing` $shareq$ => (* rule 6 *)
      let $(O_1^\bullet, \varphi_1^\bullet, S_1^\bullet) = W_{sigexp}(O, sigexp_1)$
        $(O_2^\bullet, \varphi_2^\bullet, E_2^\bullet) = W_{shareq}(O_1^\bullet + \{strid \mapsto S_1^\bullet\}, shareq)$
      in $(O_2^\bullet, \varphi_2^\bullet \circ \varphi_1^\bullet, \varphi_2^\bullet S_1^\bullet)$

$W_{prinsigexp}(O \text{ as } (A, B), sigexp) : \mathrm{Obj} \times \mathrm{Rea} \times \mathrm{Sig} =$ (* rule 7 *)
      let $(O^\bullet \text{ as } (A^\bullet, B^\bullet), \varphi^\bullet, S^\bullet) = W_{sigexp}(O, sigexp)$
        $A_0^\bullet = \mathrm{Below}(A^\bullet, \varphi^\bullet A)$
        $\Sigma^\bullet = \mathrm{Clos}_{A_0^\bullet} S^\bullet$
      in $((A_0^\bullet, B^\bullet), \varphi^\bullet, \Sigma^\bullet)$

Fig. 19. W_{sigexp} and $W_{prinsigexp}$.

we address the issue of how functor signatures give rise to nested quantification and how this affects the existence of principal signatures. Finally, in section 6.4 we give the proof of the main theorem.

6.1 Algorithm W

The algorithm for finding principal signatures appears in Figs. 19 and 20. (These are mutually recursive with $W_{funsigexp}$ concerning functor signature expressions, which we defer the presentation of until section 6.3.) There will be ample opportunity to dwell on the details of the algorithm when we prove the main theorem. For now, let us focus on one particularly important part, namely the definition of $W_{prinsigexp}$ in Fig. 19. This is the function that 'implements' rule 7. The notation 'O as (A, B)' is borrowed from Standard ML and is used when we want to introduce a variable O and simultaneously introduce variables for the components of O.

Referring to the definition of $W_{prinsigexp}$, assume that

$$(O^\bullet \text{ as } (A^\bullet, B^\bullet), \varphi^\bullet, S^\bullet) = W_{sigexp}(O, sigexp)$$

The basic idea is that we will then have $O \xrightarrow{\varphi^\bullet} O^\bullet$ and $O^\bullet \vdash sigexp \Rightarrow S^\bullet$. Here $\varphi^\bullet$ is a realization which may act on flexible names in O in order to satisfy sharing equations in *sigexp*. The processing of *sigexp* may also reveal hitherto unseen components of flexible structures in O. These will be present in $A^\bullet$. This is one reason why we do not in general have $\varphi^\bullet(A) = A^\bullet$ but rather $A^\bullet \sqsupseteq \varphi^\bullet(A)$. (Another reason is that for

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

$W_{atspec}(O \text{ as } (A,B), atspec)$: Obj × Rea × Env =
case *atspec* of
  `structure` *strid* : *sigexp* => (* rule 9 *)
    let $(O^\bullet, \varphi_1^\bullet, S^\bullet) = W_{sigexp}(O, sigexp)$
    in $(O^\bullet, \varphi_1^\bullet, \{strid \mapsto S^\bullet\})$
  | `functor` *funid* : *funsigexp* => (* rule 10 *)
    let $(O^\bullet, \varphi_1^\bullet, \Phi^\bullet) = W_{funsigexp}(O, funsigexp)$
    in $(O^\bullet, \varphi_1^\bullet, \{funid \mapsto \Phi^\bullet\})$
  | `sharing` *shareq* => (* rule 11 *)
    $W_{shareq}(O, shareq)$
  | `local` $spec_1$ `in` $spec_2$ `end` => (* rule 12 *)
    let $(O_1^\bullet \text{ as } (A_1^\bullet, B_1^\bullet), \varphi_1^\bullet, E_1^\bullet) = W_{spec}(O, spec_1)$
      $(O_2^\bullet, \varphi_2^\bullet, E_2^\bullet) = W_{spec}((A_1^\bullet, B_1^\bullet + E_1^\bullet), spec_2)$
    in $((A \text{ of } O_2^\bullet, \varphi_2^\bullet B_1^\bullet), \varphi_2^\bullet \circ \varphi_1^\bullet, E_2^\bullet)$

$W_{spec}(O \text{ as } (A,B), spec)$: Obj × Rea × Env =
case *spec* of
  empty => $(O, \mathrm{Id}, \{\})$ (* rule 13 *)
  | $atspec_1$ ⟨;⟩ $spec_2$ => (* rule 14 *)
    let $(O_1^\bullet \text{ as } (A_1^\bullet, B_1^\bullet), \varphi_1^\bullet, E_1^\bullet) = W_{atspec}(O, atspec_1)$
      $(O_2^\bullet, \varphi_2^\bullet, E_2^\bullet) = W_{spec}((A_1^\bullet, B_1^\bullet + E_1^\bullet), spec_2)$
    in $((A \text{ of } O_2^\bullet, \varphi_2^\bullet B_1^\bullet), \varphi_2^\bullet \circ \varphi_1^\bullet, \varphi_2^\bullet E_1^\bullet + E_2^\bullet)$

$W_{shareq}(O \text{ as } (A,B), shareq)$: Obj × Rea × Env =
case *shareq* of
  $longstrid_1 = longstrid_2$ => (* rule 15 *)
    let $(m_1, E_1) = B(longstrid_1)$
        fail if $longstrid_1 \notin \mathrm{Dom}(B)$
      $(m_2, E_2) = B(longstrid_2)$
        fail if $longstrid_2 \notin \mathrm{Dom}(B)$
      $\varphi^\bullet = Unify(A, N \text{ of } B, (m_1, m_2))$
    in $(\varphi^\bullet O, \varphi^\bullet, \{\})$

Fig. 20. W_{atspec}, W_{spec} and W_{shareq}.

$O^\bullet \vdash sigexp \Rightarrow S^\bullet$ to hold, $A^\bullet$ must cover $S^\bullet$, which may contain 'new' names.) This is summed up by writing $O \xrightarrow{\varphi^\bullet} O^\bullet$.

Moreover, the morphism $O \xrightarrow{\varphi^\bullet} O^\bullet$ is the best possible, in a sense which will be made precise by the main theorem. Informally, $\varphi^\bullet$ makes as few identifications of names as necessary and $O^\bullet$ is as small as possible subject to the limitation that it has to cover $S^\bullet$ and $B^\bullet$.

It is important to understand the nature of the set $(\mathrm{names}(A^\bullet) \setminus \mathrm{names}(\varphi^\bullet A))$. Let m' be a name in this set; even though m' is not free in $\varphi^\bullet(A)$, it may be the name of some substructure of a structure whose name, m, *is* free in $\varphi^\bullet(A)$.

Example 6.1 Let $S = (m, \{\})$, let $B = (\emptyset, \{\}, \{\mathtt{P} \mapsto S\})$ and let $A = [S]$. This situation might arise when the algorithm has processed the specification of `P` in Fig. 21. Let $S' = (m, \{\mathtt{Q} \mapsto (m', \{\})\})$, let $B^\bullet = B$ and $A^\bullet = (S', A)$. Just at the point where W_{sigexp} has processed the signature expression with which we specify `Y` in Fig. 21, we may

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

```
stucture P: sig end
structure Y:
  sig
    structure P' : sig structure Q: sig end end
    sharing P' = P
  end
```

Fig. 21. In the algorithm, a sharing specification can expand flexible structures in the assembly.

have $\varphi^\bullet$ = Id and $O^\bullet = (A^\bullet, B^\bullet)$. In this situation, we have that m' is not free in $\varphi^\bullet(A) = A$ but in $A^\bullet$, m' occurs below the name m, which *is* free in $\varphi^\bullet(A)$. □

In such cases, the algorithm has discovered a hitherto unseen component of a flexible structure. Otherwise, i.e. if m' does not occur in $A^\bullet$ 'below' any name which is free in $\varphi^\bullet(A)$, then m' is so to speak 'generic', i.e. m' can be quantified by a nameset prefix.

This partitioning of the name set names$(A^\bullet) \setminus$ names$(\varphi^\bullet A)$ into two sets (the nongeneric *versus* the generic ones) is seen clearly in $W_{prinsigexp}$ in the two lines:

$$A_0^\bullet = \text{Below}(A^\bullet, \varphi^\bullet A) \tag{22}$$

$$\Sigma^\bullet = \text{Clos}_{A_0^\bullet} S^\bullet \tag{23}$$

In (22) we let $A_0^\bullet$ be that part of $A^\bullet$ which is reachable in $A^\bullet$ starting from any name which occurs free in $\varphi^\bullet(A)$. The names that occur free in $A_0^\bullet$ must not be quantified. In (23) we then quantify the remaining names in order to form the signature $\Sigma^\bullet$. Notice that $W_{prinsigexp}$ returns the assembly $A_0^\bullet$ (not $A^\bullet$), i.e. the algorithm 'discharges' that part of the assembly which has just been quantified.

Below we define the Below and Clos operations and show that $\Sigma^\bullet$ by construction automatically is well-formed. Also, we prove, for example, that $A_0^\bullet \trianglelefteq A^\bullet$ holds, so that $A_0^\bullet$ and $A^\bullet$ can play the parts of A and A', respectively, in the definition of principal signature.

Let A be an admissible semantic object and let N be a name set. The names below N in A are the names that are reachable in Graph(A), starting from a node whose name is in N. Put differently, the *names below* N *in* A, written below(A, N), is the least set N' satisfying

1. $N \cap \text{names}(A) \subseteq N'$
2. Whenever (m, E) occurs free in A and $m \in N'$ then $\text{names}(\text{skel}(E)) \subseteq N'$

The *structures below* N *in* A, written Below(A, N), is defined by

$$\text{Below}(A, N) = \{\text{skel}(m, E) \mid (m, E) \text{ occurs free in } A \text{ and } m \in \text{below}(A, N)\}$$

When A' is a semantic object, we write below(A, A') for below$(A, \text{names}(A'))$, and we write Below(A, A') as an abbreviation of Below$(A, \text{names}(A'))$. Also, we shall identify the set Below(A, N) with any assembly $[S_1, \ldots, S_n]$, where $\{S_1, \ldots, S_n\} = \text{Below}(A, N)$.

One easily proves that $\text{below}(A, N) = \text{names}(\text{Below}(A, N))$, for all admissible semantic objects A and name sets N.

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

Lemma 6.1
Let A be an admissible object and let N be a name set. Then $\mathrm{Below}(A, N) \trianglelefteq A$.

Proof
Certainly, $\mathrm{Below}(A, N) \sqsubseteq A$. It remains to show that $\mathrm{Below}(A, N)$ covers A on $\mathrm{below}(A, N)$. But that follows from the definitions of Below and skel, in particular from the fact that the skel function does not 'throw away' functor components — it merely replaces them by a closed functor signature, cf. section 3.3. □

Next we show an important lemma, which concerns the following situation:

$$\begin{array}{ccc} (A_1, B_1) & \xrightarrow{\varphi} & (A_2, B_2) \\ \triangledown\vert & & \triangledown\vert \\ (A_1', B_1) & \xrightarrow{\varphi} & (\mathrm{Below}(A_2, \varphi A_1'), B_2) \end{array}$$

The lemma says that the bottom morphism exists if the other parts of the diagram are given, provided that $A_1' \trianglelefteq A_1$ (as we shall see in the proof, $A_1' \sqsubseteq A_1$ would not be enough here).

Lemma 6.2
Assume $B_1 \sqsubseteq A_1' \trianglelefteq A_1$ and $(A_1, B_1) \xrightarrow{\varphi} (A_2, B_2)$ and let
$A_2' = \mathrm{Below}(A_2, \varphi A_1')$. Then $(A_1', B_1) \xrightarrow{\varphi} (A_2', B_2)$.

Proof
Since $A_1' \sqsubseteq A_1$ and φA_1 is admissible we have $\varphi A_1' \sqsubseteq \varphi A_1$, by Lemma 5.3. Since $\varphi A_1 \sqsubseteq A_2$, we then have $\varphi A_1' \sqsubseteq A_2$. Thus

$$A_2' = \mathrm{Below}(A_2, \varphi A_1') \sqsupseteq \varphi A_1' \tag{24}$$

Since by assumption $A_1' \sqsupseteq B_1$, we have $(A_1', B_1) \in \mathrm{Obj}$. To see that $(A_2', B_2) \in \mathrm{Obj}$, note that $A_1' \sqsupseteq B_1$ implies $\varphi A_1' \sqsupseteq \varphi B_1 = B_2$, by Lemma 5.3; thus $A_2' \sqsupseteq B_2$, by (24). Let us show that φ is a morphism from (A_1', B_1) to (A_2', B_2). By assumption φ is fixed on N of B_1 and $\varphi(B_1) = B_2$, as desired. Also $\varphi(A_1') \sqsubseteq A_2'$ by (24), as desired. Let $N_1 = N$ of B_1. It remains to prove that A_1' covers A_2' on N_1. We have $N_1 \cap \mathrm{names}(A_2') \subseteq \mathrm{names}(A_1')$, as required (since $A_1' \sqsupseteq B_1$). Let (m, E_2') be a structure occurring free in A_2' with $m \in N_1$ and assume $id \in \mathrm{Dom}(E_2')$. By the definition of A_2' there exists E_2 such that (m, E_2) occurs free in A_2 and $id \in \mathrm{Dom}(E_2)$. Since A_1 covers A_2 on N_1 there exists an E_1 such that (m, E_1) occurs free in A_1 and $id \in \mathrm{Dom}(E_1)$. But then, since A_1 is a *conservative* cover of A_1' and $m \in \mathrm{names}(A_1')$, there exists an E_1' such that (m, E_1') occurs free in A_1' and $id \in \mathrm{Dom}(E_1')$, as desired. □

There is a characterization of $\mathrm{below}(A, N)$ which often is useful in proofs. Recall that StrName is the set of all structure names. Let $\mathscr{P}(\mathrm{StrName})$ denote the set of subsets of StrName. Given A and N, let $\mathscr{F}_{A,N} : \mathscr{P}(\mathrm{StrName}) \to \mathscr{P}(\mathrm{StrName})$ be defined by

$$\mathscr{F}_{A,N}(N') = \{m \in \mathrm{names}(A) \mid m \in N \text{ or there exists an } (m', E') \text{ which occurs free in } A \text{ and satisfies that } m' \in N' \text{ and } m \in \mathrm{names}(\mathrm{skel}(E'))\}$$

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

Then $\mathscr{F}_{A,N}$ is continuous with respect to set inclusion and its least fixed point is exactly below(A, N):

$$\text{below}(A,N) = \bigcup_{i \geq 0} \mathscr{F}^i_{A,N}(\emptyset)$$

An example of the use of this property is the proof of the next lemma. A morphism φ in K is an *epimorphism* if for every pair ψ_1, ψ_2 of morphisms in K, if $\psi_1 \circ \varphi = \psi_2 \circ \varphi$ then $\psi_1 = \psi_2$.

Lemma 6.3
Let $(A_1, B_1) = O_1$ and $(A_2, B_2) = O_2$ and assume $O_1 \xrightarrow{\varphi} O_2$.
Then $O_1 \xrightarrow{\varphi}(\text{Below}(A_2, \varphi A_1), B_2)$ is an epimorphism.

Proof
Let $A'_2 = \text{Below}(A_2, \varphi A_1)$ and $O'_2 = (A'_2, B_2)$. To prove that $O_1 \xrightarrow{\varphi} O'_2$ is an epimorphism, let O_3 be an object in K and ψ_1 and ψ_2 be realizations such that

$$O_1 \xrightarrow{\varphi} O'_2 \overset{\psi_1}{\underset{\psi_2}{\rightrightarrows}} O_3$$

commutes. We wish to prove that $\psi_1(m) = \psi_2(m)$, for all $m \in \text{names}(O'_2)$. By the definition of A'_2, this amounts to proving that

$$\forall i \geq 0 \forall m \in \mathscr{F}^i_{(A_2, \varphi A_1)}(\emptyset).\psi_1(m) = \psi_2(m)$$

where $\mathscr{F}$ was defined above. This, however, is easily shown by induction on i, using that $(\psi_1 \circ \varphi)(m) = (\psi_2 \circ \varphi)(m)$, for all $m \in \text{names}(O_1)$, that $\psi_1(A'_2) \sqsubseteq A_3$ and $\psi_2(A'_2) \sqsubseteq A_3$ and that A_3 is consistent, where $A_3 = A$ of O_3. □

This finishes the treatment of the Below operation. Now let us look at the Clos operation and its properties.

For all structures S and name sets N, we define $\text{Clos}_N S$ to be the signature $(N')S$, where $N' = \text{names}(S) \setminus N$. For every semantic object A, $\text{Clos}_A S$ means $\text{Clos}_{\text{names}(A)} S$.

Lemma 6.4 (Closure and well-formedness, Version 1)
For all name sets N, assemblies A and structures S, if $A \sqsupseteq S$ then $\text{Clos}_{\text{Below}(A,N)} S$ is a well-formed signature.

Proof
Since $A \sqsupseteq S$, S is admissible and in particular well-formed, as required. Write $\text{Clos}_{\text{Below}(A,N)} S$ in the form $(N')S$, i.e. let N' be $\text{names}(S) \setminus \text{below}(A, N)$. Let (m, E) be a structure occurring free in S and assume $m \notin N'$. Then $m \in \text{below}(A, N)$. Since $A \sqsupseteq S$, we have $A \sqsupseteq (m, E)$. Since $m \in \text{below}(A, N)$ we therefore have $\text{names}(\text{skel}(E)) \subseteq \text{below}(A, N)$. Hence $\text{names}(\text{skel}(E)) \cap N' = \emptyset$, as required. □

Lemma 6.5 (Closure and well-formedness, Version 2)
For all assemblies A and A' and all structures S, if $A \trianglelefteq A'$ and $A' \sqsupseteq S$ then $\text{Clos}_A S$ is a well-formed signature.

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

Proof

Follows directly from Lemma 6.4 and the observation that $A \trianglelefteq A'$ implies below$(A', A) = $ names(A). $\square$

If A' is a conservative cover of A then the names by which A' extends A can be renamed without affecting the fact that A' is a conservative cover of A:

Lemma 6.6

Let $A \trianglelefteq A'$ and let $N = $ names$(A') \setminus$ names(A). Let φ be any realization which is injective on N, is fixed on names(A) and satisfies Ran$(\varphi) \cap$ names$(A) = \emptyset$. Then $A \trianglelefteq \varphi(A')$.

Proof

Simple verification. $\square$

Finally, we prove that a signature can only be principal if it does not contain free names which could be quantified:

Lemma 6.7 (Closure and principality)

Let $O = (A, B)$ be an object in K. For all name sets N and structures S, if $N \cap$ names$(O) = \emptyset$ and $N \subseteq$ names(S) and $(N)S$ is principal for *sigexp* in O then $(N)S = \mathrm{Clos}_A S$.

Proof

We know that $N \subseteq$ names(S). Since $N \cap$names$(O) = \emptyset$, we know that $N \cap$names$(A) = \emptyset$. To show $(N)S = \mathrm{Clos}_A S$, it just remains to show names$((N)S) \subseteq$ names(A). Since $(N)S$ is principal for *sigexp* in (A, B) there exists an A' such that $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S$, and

$$\text{For all } O',\ \varphi' \text{ and } S',\ \text{if } O \xrightarrow{\varphi'} O' \text{ and } O' \vdash sigexp \Rightarrow S' \text{ then } \varphi'((N)S) \geq S' \quad (25)$$

In particular, for $\varphi' = \mathrm{Id}$, (25) states that

$$\text{For all } A'' \text{ and } S',\ \text{if } A \sqsubseteq A'' \text{ and } A \text{ covers } A'' \text{ on } N \text{ of } B \text{ and } A'', B \vdash sigexp \Rightarrow S' \text{ then } (N)S \geq S' \quad (26)$$

Let φ be an injective realization which maps names in the set names$(A') \setminus$ names(A) to distinct fresh names and is the identity on all other names. By Lemma 6.6 we have that $A \trianglelefteq \varphi A'$. Since $A \sqsupseteq B$, we then have $O' \xrightarrow{\varphi} O''$, where $O' = (A', B)$ and $O'' = (\varphi A', B)$. Since $O' \vdash sigexp \Rightarrow S$ we then have $O'' \vdash sigexp \Rightarrow \varphi S$, by Theorem 5.1. Thus by (26) we have $(N)S \geq \varphi S$. But this implies names$((N)S) \subseteq$ names(A), as required. $\square$

6.2 *Statement of the main theorem*

We now state a stronger version of Theorem 3.1, suitable for inductive proof. We refer to this theorem as the *main theorem*:

Theorem 6.1 (Main theorem)

Let *phrase* be one of *sigexp*, *spec*, *atspec*, *shareq* or *funsigexp* and let $O \in \mathrm{Obj}$. If there exists some O', φ and P' such that $O \xrightarrow{\varphi} O'$ and $O' \vdash phrase \Rightarrow P'$ then $(O^*, \varphi^*, P^*) = W(O, phrase)$ succeeds and

1. $O \xrightarrow{\varphi^*} O^*$ and $O^* \vdash phrase \Rightarrow P^*$
2. For all O' , φ' and P satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash phrase \Rightarrow P'$ there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{27}$$

   commutes and $\psi P^* = P'$.

Readers who are familiar with the completeness result for the Damas-Milner algorithm W (Damas and Milner, 1982) will recognize the overall structure of the above theorem. We have not proved that W fails if there exists no O', φ and P' with $O \xrightarrow{\varphi} O'$ and $O' \vdash phrase \Rightarrow P'$; we believe this is the case, but we do not need this result to prove Theorem 3.1.

It is not clear, of course, that the above theorem really implies Theorem 3.1. We will spend the rest of this section demonstrating this. In the process, we shall prove a number of lemmas, which we also shall use in the proof of Theorem 6.1.

Write O, O' and O^* in the form (A, B), (A', B') and (A^*, B^*), respectively. In section 6.1 we considered the special case where *phrase* is *sigexp* and P, P' and P^* are structures S, S', S^*, respectively. We saw that the assembly and signature we are interested in are $A_0^* = \mathrm{Below}(A^*, \varphi^*(A))$ and $\Sigma^* = \mathrm{Clos}_{A_0^*} S^*$. More generally, in order to separate that part of the diagram (27) which is 'generic' from that part which is not, we shall often need to derive from diagrams like (27) another diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O_0^* \arrow[rr, "\psi"] & & O'' \end{tikzcd} \tag{28}$$

where the realizations are the same, but $O_0^* = (\mathrm{Below}(A^*, \varphi^*(A)), B^*)$ and $O'' = (A'', B')$ for some suitable A''. Below we prove a lemma which gives conditions that are sufficient to ensure that this separation really is possible. As a first step, let us prove the following lemma:

Lemma 6.8 (Cover)

Let φ be a realization which is fixed on N. For all assemblies A_1 and A_2, if A_1 covers φA_2 on N then A_1 covers A_2 on N.

Proof

Let (m, E) occur free in A_2, $m \in N$ and $id \in \mathrm{Dom}\, E$, where $id \in \mathrm{StrId} \cup \mathrm{FunId}$. Then, since φ is fixed on N, the structure $S' = \varphi(m, E) = (m, \varphi E)$ occurs free in φA_2 and

$m \in N$. Since A_1 covers φA_2 on N there exists an E_0 such that (m, E_0) occurs free in A_1 and $id \in \operatorname{Dom} E_0$, as required. $\square$

The next step is to prove that decomposition of realization preserves cover.

Lemma 6.9

If $\varphi_1 A_1 \sqsubseteq A_2$ and $\varphi_2 A_2 \sqsubseteq A_3$ and A_1 covers A_3 on N and φ_1 and φ_2 are fixed on N, then A_1 covers A_2 on N and A_2 covers A_3 on N.

Proof

As A_1 covers A_3 on N and $\varphi_2 A_2 \sqsubseteq A_3$, A_1 covers φA_2 on N. Thus A_1 covers A_2 on N, by Lemma 6.8.

To prove that A_2 covers A_3 on N, let (m, E) be a structure which occurs free in A_3 and let id be a structure or functor identifier in the domain of E. Assume $m \in N$. Since A_1 covers A_3 on N there exists an E_1 such that (m, E_1) occurs free in A_1 and $id \in \operatorname{Dom} E_1$. Then, since φ_1 is fixed on N and $\varphi_1 A_1 \sqsubseteq A_2$, $\varphi(m, E_1) = (m, \varphi E_1)$ is covered by A_2. Thus there exists an E_2 such that (m, E_2) occurs free in A_2 and $id \in \operatorname{Dom} E_2$. Thus A_2 covers A_3 on N. $\square$

Finally, we can state and prove the promised lemma about the existence of (28):

Lemma 6.10

Let $O = (A, B)$, $O' = (A', B')$ and $O^\bullet = (A^\bullet, B^\bullet)$ be objects in K and assume that the diagram (a) below commutes. Let $A_0^\bullet = \text{Below}(A^\bullet, \varphi^\bullet(A))$, let $O_0^\bullet = (A_0^\bullet, B^\bullet)$, let A_0' be any assembly satisfying $\text{Below}(A', \varphi(A)) \sqsubseteq A_0' \sqsubseteq A'$ and let $O_0' = (A_0', B')$. Then the diagram (b) exists in K and commutes.

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd} \qquad\qquad \begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O_0^\bullet \arrow[rr, "\psi"] & & O_0' \end{tikzcd}$$

(a) (b)

Proof

That $O^\bullet \xrightarrow{\varphi^\bullet} O_0^\bullet$ is a morphism in K follows from Lemmas 6.1 and 6.2. Similarly, $O \xrightarrow{\varphi} (\text{Below}(A', \varphi(A)), B')$ is a morphism in K by Lemmas 6.1 and 6.2. Since A covers A' on N of $B (= N$ of $B')$ and $A_0' \sqsubseteq A'$ we have that A covers A_0' on N of B. Thus $O \xrightarrow{\varphi} O_0'$ is a morphism in K. The real question is whether the bottom morphism exists. (If it exists, then (b) commutes, since (a) commutes.) Referring to the definition of K, we certainly have that ψ is fixed on N of $B^\bullet$ and that $\psi(B^\bullet) = B'$. Moreover, $\psi A_0^\bullet = \psi(\text{Below}(A^\bullet, \varphi^\bullet(A))) \sqsubseteq \text{Below}(A', \psi(\varphi^\bullet(A)))$ since the morphisms $O \xrightarrow{\varphi^\bullet} O^\bullet \xrightarrow{\psi} O'$ exist. Thus, since (a) commutes, we have $\psi A_0^\bullet \sqsubseteq \text{Below}(A', \varphi A) \sqsubseteq A_0'$, so $\psi(A_0^\bullet) \sqsubseteq A_0'$, as required. It remains to prove that $A_0^\bullet$ covers A_0' on N of B. We have $\varphi^\bullet(A) \sqsubseteq A_0^\bullet$ and $\psi A_0^\bullet \sqsubseteq A_0'$ and A covers A_0' on N of B' and $\varphi^\bullet$ and ψ are fixed on N of B — since (a) exists. But then Lemma 6.9 gives that $A_0^\bullet$ covers A_0' on N of B, as required. $\square$

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

We can now prove that for all O and *phrase* if one can find $O^\bullet$, $\varphi^\bullet$ and $P^\bullet$ satisfying (1) and (2) of Theorem 6.1, then one thereby has a way of obtaining principal signatures:

Lemma 6.11 (The main theorem gives principal signatures)

Let $O \in \mathrm{Obj}$ and let *phrase* be a signature expression. Assume that object $O^\bullet$, realization $\varphi^\bullet$ and structure $P^\bullet$ satisfy (1) and (2) of Theorem 6.1. Let $(A^\bullet, B^\bullet) = O^\bullet$, $A_0^\bullet = \mathrm{Below}(A^\bullet, \varphi^\bullet(A))$ and $\Sigma^\bullet = \mathrm{Clos}_{A_0^\bullet} P^\bullet$. Then $\Sigma^\bullet$ is principal for *phrase* in $(A_0^\bullet, B^\bullet)$.

This is certainly good news, for the $\Sigma^\bullet$ constructed above is precisely the $\Sigma^\bullet$ which $W_{prinsigexp}$ produces (see Fig. 19). Of course, the above lemma does not state that $\Sigma^\bullet$ is principal for the signature expression in O, but once we have proved Lemma 6.11, it is easy to prove that Theorem 6.1 implies Theorem 3.1. We now prove Lemma 6.11:

Proof

Let *sigexp* = *phrase* and $S^\bullet = P^\bullet$. Let $O_0^\bullet = (A_0^\bullet, B^\bullet)$. Following the definition of principal signature, it suffices to prove

$$O_0^\bullet \in \mathrm{Obj} \tag{29}$$

$$A_0^\bullet \trianglelefteq A^\bullet \quad \text{and} \quad A^\bullet, B^\bullet \vdash \mathit{sigexp} \Rightarrow S^\bullet \tag{30}$$

$$\text{For all } \varphi,\ O' \text{ and } S',\ \text{if } O_0^\bullet \xrightarrow{\varphi} O' \text{ and } O' \vdash \mathit{sigexp} \Rightarrow S' \text{ then } \varphi(\Sigma^\bullet) \geq S' \tag{31}$$

Since $O \xrightarrow{\varphi^\bullet} O^\bullet$ we have $O \xrightarrow{\varphi^\bullet} O_0^\bullet$ by Lemma 6.2. Thus (29) holds. Also, (30) follows from Lemma 6.1 and the assumption $O^\bullet \vdash \mathit{sigexp} \Rightarrow S^\bullet$. Let us now prove (31). Let φ, O' and S' be such that $O_0^\bullet \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{sigexp} \Rightarrow S'$. Clearly, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi \circ \varphi^\bullet"] & \\ O_0^\bullet \arrow[rr, "\varphi"] & & O' \end{tikzcd} \tag{32}$$

commutes. By assumption, there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi \circ \varphi^\bullet"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{33}$$

commutes and $\psi S^\bullet = S'$. To prove $\varphi(\Sigma^\bullet) \geq S'$ it will therefore suffice to prove that $\varphi(n) = \psi(n)$, for all $n \in \mathrm{names}(\Sigma^\bullet)$ — for if so, ψ both performs the realization of the free names in $\Sigma^\bullet$ and the instantiation of the bound names of $\Sigma^\bullet$. By the definition of $\Sigma^\bullet$ we have $\mathrm{names}(\Sigma^\bullet) \subseteq \mathrm{names}(A_0^\bullet)$. Let us prove

$$\varphi(m) = \psi(m), \text{ for all } m \in \mathrm{names}(A_0^\bullet) \tag{34}$$

Since (33) commutes, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi \circ \varphi^*"] & \\ O_0^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{35}$$

commutes by Lemma 6.10. By Lemma 6.3 we have that $O \xrightarrow{\varphi^*} O_0^*$ is an epimorphism. But then, since (32) and (35) commute, we have (34), as required. Thus Σ^* is principal for *sigexp* in O_0^*. □

Lemma 6.12

Theorem 6.1 implies Theorem 3.1

Proof

Let B be a basis, A an assembly, $A \sqsupseteq B$, $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S'$, for some S'. Let $O = (A, B)$ and $O' = (A', B)$. Then $O \xrightarrow{\mathrm{Id}} O'$ and $O' \vdash sigexp \Rightarrow S'$. By Theorem 6.1, $(O^*, \varphi^*, S^*) = W_{sigexp}(O, sigexp)$ succeeds and $O \xrightarrow{\varphi^*} O^*$ and $O^* \vdash sigexp \Rightarrow S^*$. Also by Theorem 6.1, there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\mathrm{Id}"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

commutes and $\psi S^* = S'$. Let $(A^*, B^*) = O^*$. Let $A_0^* = \mathrm{Below}(A^*, \varphi^*(A))$ and let $O_0^* = (A_0^*, B^*)$. Since $A \trianglelefteq A'$ we have $\mathrm{Below}(A', \mathrm{Id}(A)) \sim A$. Thus by Lemma 6.10 the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\mathrm{Id}"] & \\ O_0^* \arrow[rr, "\psi"] & & O \end{tikzcd}$$

exists in K and commutes. By Lemma 6.11 we have that $\Sigma^* = \mathrm{Clos}_{A_0^*} S^*$ is a principal signature for *sigexp* in O_0^*. Thus $O_0^* \vdash sigexp \Rightarrow \Sigma^*$. Since $O_0^* \xrightarrow{\psi} O$ we have $O \vdash sigexp \Rightarrow \psi\Sigma^*$ by Theorem 5.1. In particular, $\psi\Sigma^*$ is principal for *sigexp* in O. □

We are thus left with proving the main theorem itself. Before doing this, let us consider the most interesting part of W in isolation, the part concerning functor signature expressions.

6.3 Functor signatures

Our type discipline for HML admits functor signatures that occur inside signatures. Since functor signatures take the form $\Phi = (N)(S, (N')S')$ and signatures take the form $(N_0)S_0$ (where Φ can be a component of S_0), the type discipline for higher-order modules involves nested quantification. In this respect, our type discipline is a departure from the well-known principle of ML-style polymorphism that quantification is at the outermost level only.

It is actually rather surprising, at least at first sight, that the principality theorem (Theorem 3.1) holds. For a signature $\Sigma_0 = (N_0)S_0$ to be principal for some signature expression it must be the case that all other possible results of elaborating the signature expression are instances of Σ_0, in the sense defined in Section 3.5. But instantiation only admits realization of the outermost bound names, i.e. those in N_0. In other words, for the principality theorem to hold, the inner quantifications must be the same in every possible elaboration. Fortunately, this can be achieved by demanding that the argument and result signature expressions in functor signature expressions be elaborated to *principal* signatures. To be more specific, consider a functor signature expression

$$(strid : sigexp_1)\ \ sigexp_2$$

to be elaborated in (A, B), say. The functor signature expression may occur deep inside some signature declaration, so the assembly A and basis B may contain flexible structures. Assume we can elaborate $sigexp_1$ to Σ_1, a principal signature for $sigexp_1$ in A, B. Now the way possible elaborations can vary is essentially just by the choice of names of flexible structures, subject to the requirements about admissibility and cover. Thus one has to consider what $sigexp_1$ would elaborate to in (A', B'), if $(A, B) \xrightarrow{\varphi} (A', B')$, for some realization φ. However, we know that $\varphi\Sigma_1$ will be principal for $sigexp_1$ in (A', B') — cf. the proof of Theorem 5.1. Notice that applying a realization to a signature only affects the free names (although it may require renaming of the bound names), so the nameset prefix is essentially the same in all possible elaborations.

Thus we see that for the principality theorem to hold, it is crucial that principality is preserved under realization. But it also has to be preserved under 'inverse' realization: if there exists *some* φ, O' and Σ' with $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow \Sigma'$ and if $(O^*, \varphi^*, \Sigma^*)$ is the result of $W(O, sigexp)$, then Σ^* had better be principal for $sigexp$ in O^*.

Thus the situation is somewhat more involved in the higher-order language than in Standard ML, where signature expressions are only ever elaborated in 'rigid' bases (compare with Theorems 11.4 and 11.5 of the Commentary — Milner and Tofte, 1991); indeed, the changes we have made relative to the Standard ML semantics are mostly motivated by the need to obtain good interaction between realization and principality.

We shall now prove two lemmas that concern precisely the interaction between realization and principality. The first one will be used in the proof of Theorem 6.1 in the case where we have assumed $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow (N')S'$. By induction we will have obtained (O^*, φ^*, S^*) and ψ such that $O^* \vdash sigexp \Rightarrow S^*$ and $\psi(S^*) = S'$. We now wish to infer $\psi((N^*)S^*) = (N')S'$, where N^* is a suitable nameset prefix. But this only holds because $(N')S'$ has to be principal:

$W_{funsigexp}(O \text{ as } (A, B), funsigexp) : \text{Obj} \times \text{Rea} \times \text{FunSig} =$
case *funsigexp* of
  $(strid\!:\!sigexp_1)\,sigexp_2$ => (* rule 8 *)
  let $(O_1^\bullet \text{ as } (A_1^\bullet, B_1^\bullet), \varphi_1^\bullet, \Sigma_1^\bullet) = W_{prinsigexp}(O, sigexp_1)$
    $(N_1^\bullet)S_1^\bullet = \Sigma_1^\bullet$, where all names in $N_1^\bullet$ are new
    $O_2 = ((S_1^\bullet, A_1^\bullet), B_1^\bullet + N_1^\bullet + \{strid \mapsto S_1^\bullet\})$
    $(O_2^\bullet, \varphi_2^\bullet, \Sigma_2^\bullet) = W_{prinsigexp}(O_2, sigexp_2)$
    $A_2^\bullet = A \text{ of } O_2^\bullet$
    $A^\bullet = \text{Below}(A_2^\bullet, \varphi_2^\bullet(A_1^\bullet))$
    $\varphi^\bullet = \varphi_2^\bullet \circ \varphi_1^\bullet$
    $\Phi^\bullet = (N_1^\bullet)(\varphi_2^\bullet S_1^\bullet, \Sigma_2^\bullet)$
  in if $N_1^\bullet \cap \text{names}(A^\bullet) = \emptyset$ then $((A^\bullet, \varphi_2^\bullet B_1^\bullet), \varphi^\bullet, \Phi^\bullet)$
    else **fail**

Fig. 22. $W_{funsigexp}$.

Lemma 6.13 (Realization and principality)
Let $O_i = (A_i, B_i)$ be objects in K, for $i = 1, 2$, let φ be a realization and assume $O_1 \xrightarrow{\varphi} O_2$. Further let S_1 and S_2 be structures, N_1 and N_2 be name sets. If $N_i \cap \text{names}(A_i) = \emptyset$ and $N_i \subseteq \text{names}(S_i)$ and $(N_i)S_i$ is principal for *sigexp* in O_i ($i = 1, 2$), and $\varphi S_1 = S_2$ then $\varphi((N_1)S_1) = (N_2)S_2$.

Proof
We wish to prove that φ maps bound names to bound names:

$$\varphi \text{ is injective on } N_1 \text{ and } \varphi N_1 \subseteq N_2 \tag{36}$$

without capture of names:

$$N_2 \cap \varphi(\text{names}((N_1)S_1)) = \emptyset \tag{37}$$

Since $(N_i)S_i$ is principal for *sigexp* in O_i, there exists an A_i' ($i = 1, 2$) such that $A_i \trianglelefteq A_i'$ and $A_i', B \vdash sigexp \Rightarrow S_i$. Let φ' be a realization which satisfies $\varphi'(n) = \varphi(n)$, for all $n \in \text{names}(A_1)$ but in addition maps all names in the set $\text{names}(A_1') \setminus \text{names}(A_1)$ to distinct fresh names. By Lemma 6.7 we have $(N_i)S_i = \text{Clos}_{A_i} S_i$, $i = 1, 2$. In particular, $N_1 \subseteq \text{names}(A_1') \setminus \text{names}(A_1)$, so

$$\varphi' \text{ maps the names in } N_1 \text{ to distinct fresh names} \tag{38}$$

Since $A_1 \trianglelefteq A_1'$ and $(A_1, B_1) \xrightarrow{\varphi} (A_2, B_2)$ there exists an assembly A_2' such that $\varphi A_1 \sqsubseteq A_2 \sqsubseteq A_2'$ and $(A_1', B_1) \xrightarrow{\varphi'} (A_2', B_2)$. Since $A_1', B_1 \vdash sigexp \Rightarrow S_1$ we then have $A_2', B_2 \vdash sigexp \Rightarrow \varphi' S_1$, by Theorem 5.1. But then, since $A_2 \sqsubseteq A_2'$ and $(N_2)(\varphi S_1)$ is principal for *sigexp* in (A_2, B_2), we have $(N_2)(\varphi S_1) \geq \varphi' S_1$. This, together with (38), gives (36). As for (37), we have $\text{names}((N_1)S_1) \subseteq \text{names}(A_1)$. Since $(N_2)(\varphi S_1) = \text{Clos}_{A_2}(\varphi S_1)$ and $\varphi A_1 \sqsubseteq A_2$ we then have (37), as desired. □

The other lemma is used in the proof of Theorem 6.1 in the case concerning functor signature expressions. The inference of $O' \vdash funsigexp \Rightarrow \Phi'$ takes the following

https://doi.org/10.1017/S0956796800001088 Published online by Cambridge University Press

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^\bullet"'] \arrow[dr, "\varphi"] & \\ O_1^\bullet \arrow[rr, "\psi_1"] & & O' \end{tikzcd}$$

(a)

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^\bullet"'] \arrow[dr, "\psi_1"] & \\ O_2^\bullet \arrow[rr, "\psi"] & & O_2' \end{tikzcd}$$

(b)

$$\begin{tikzcd} & O_1^\bullet \arrow[dl, "\varphi_2^\bullet"'] \arrow[dr, "\psi_1"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

(c)

Fig. 23. The diagrams involved in the proof case concerning functor signature expressions

form:

$$\frac{A', B' \vdash sigexp_1 \Rightarrow (N_1')S_1' \quad N_1' \cap \mathrm{names}(A') = \emptyset \qquad (S_1', A'), B' + N_1' + \{strid \mapsto S_1'\} \vdash sigexp_2 \Rightarrow (N_2')S_2'}{A', B' \rhd (strid : sigexp_1)\, sigexp_2 \Rightarrow (N_1')(S_1', (N_2')S_2')} \tag{39}$$

This is to be compared with the the algorihm $W_{funsigexp}$ in Fig. 22.

Using induction on $O \xrightarrow{\varphi} O'$ and $sigexp_1$ it will be possible to construct the diagram in Figure 23(a), where $\varphi_1^\bullet$ and $O_1^\bullet$ are as given in the algorithm. Next, the algorithm constructs O_2; let $O_2' = ((S_1', A'), B' + N_1' + \{strid \mapsto S_1'\})$, i.e. the object in the second premise of (39). It turns out that one has $O_2 \xrightarrow{\psi_1} O_2'$ (as well as $O_1^\bullet \xrightarrow{\psi_1} O'$). Thus we use induction again and get the diagram in Fig. 23(b). The crucial step is now that we want to "cut down" this diagram to the diagram in Fig. 23(c). (Note that the realizations in (b) and (c) are the same; the difference is that in (c) we have removed those structures which are quantified by $N_1^\bullet$ and N_1'.) If only this can be done, then we can paste (a) and (c) together along the morphism ψ_1 and get

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet = \varphi_2^\bullet \circ \varphi_1^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

However, *can* (c) be constructed from (b)? Can one be sure, for example, that $\varphi_2^\bullet$ does not map a name which was free in $B_1^\bullet$ to a name which is a member of $N_1^\bullet$?

A related problem is that in order to prove $A^\bullet, B^\bullet \vdash funsigexp \Rightarrow \Phi^\bullet$, where $A^\bullet$, $B^\bullet$ and $\Phi^\bullet$ are the objects constructed by W, the inference rule (8) demands that $sigexp_2$ be elaborated in the assembly $(\varphi_2^\bullet S_1^\bullet, A^\bullet)$. The induction hypothesis will give us that $sigexp_2$ elaborates in $A_2^\bullet$; but how are these two assemblies related?

The lemma below answers the above questions. The proof of the lemma is a bit draining, so it is relegated to the Appendix. In the statement of the lemma, we refer to diagrams (b) and (c) of Fig. 23.

Lemma 6.14

Let O, O' and $O_1^\bullet$ be objects in K (with $O_1^\bullet = (A_1^\bullet, B_1^\bullet)$, etc). Assume that $O_1^\bullet \xrightarrow{\psi_1} O'$ and $A_1^\bullet \sqsupseteq (N_1^\bullet)S_1^\bullet$ and $A' \sqsupseteq (N_1')S_1'$ and $N_1^\bullet = N_1'$ and $N_1' \cap (\mathrm{names}(A_1^\bullet) \cup \mathrm{names}(A')) = \emptyset$. Let $O_2 = ((S_1^\bullet, A_1^\bullet), B_1^\bullet + N_1^\bullet + \{strid \mapsto S_1^\bullet\})$ and $O_2' = ((S_1', A'), B' + N_1' + \{strid \mapsto S_1'\})$. Let $A_2^\bullet$ be an assembly satisfying $A_2^\bullet \sim \mathrm{Below}(A_2^\bullet, \varphi_2^\bullet(A \text{ of } O_2))$. Let $A^\bullet = \mathrm{Below}(A_2^\bullet, \varphi_2^\bullet(A_1^\bullet))$ and $O^\bullet = (A^\bullet, \varphi_2^\bullet B_1^\bullet)$. If (b) commutes, then $A_2^\bullet \sim (A^\bullet, \varphi_2^\bullet(S_1^\bullet))$, $N_1^\bullet \cap \mathrm{names}(A^\bullet) = \emptyset$ and the diagram (c) commutes.

6.4 *Proof of the main theorem*

This section is the proof of Theorem 6.1. The proof is by induction on the depth of inference of $O' \vdash phrase \Rightarrow P'$. There is one case for each rule. The cases for rules 6, 9–11, and 13 are all straightforward and are not included. The case for `local` (rule 12) is very similar to the case for sequential specifications (rule 14) and is therefore also omitted. We now deal with each of the remaining cases. For clarity, each case is divided into two parts, corresponding to parts *(1)* and *(2)* of Theorem 6.1.

$\boxed{\text{Rule 4, } sigexp \equiv \texttt{sig}\ spec\ \texttt{end}}$

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow (m', E')$. Then $O' \vdash spec \Rightarrow E'$. By induction, $(O_1^\bullet, \varphi_1^\bullet, E_1^\bullet) = W_{spec}(O, spec)$ succeeds and we have $O \xrightarrow{\varphi_1^\bullet} O_1^\bullet$ and $O_1^\bullet \vdash spec \Rightarrow E_1^\bullet$. Following W, let $(A_1^\bullet, B_1^\bullet) = O_1^\bullet$ and let $m^\bullet$ be a fresh structure name. Let $A^\bullet = ((m^\bullet, E_1^\bullet), A_1^\bullet)$, $O^\bullet = (A^\bullet, B_1^\bullet)$, $\varphi^\bullet = \varphi_1^\bullet$ and $S^\bullet = (m^\bullet, E_1^\bullet)$. Note that $O^\bullet$ is admissible because $m^\bullet$ is fresh. By $O \xrightarrow{\varphi_1^\bullet} O_1^\bullet$ and the definition of $A^\bullet$ we have $O \xrightarrow{\varphi^\bullet} O^\bullet$ as required. From $O_1^\bullet \vdash spec \Rightarrow E_1^\bullet$ and $O_1^\bullet \xrightarrow{\mathrm{Id}} O^\bullet$ we get $O^\bullet \vdash spec \Rightarrow E_1^\bullet$, by Theorem 5.1. Thus $O^\bullet \rhd \texttt{sig}\ spec\ \texttt{end} \Rightarrow S^\bullet$, by rule 4, and since $O^\bullet \sqsupseteq S^\bullet$ we then have $O^\bullet \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow S^\bullet$, as desired.

Part 2: Let O', φ and S' be objects satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow S'$. Let $(m', E') = S'$. By induction there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^\bullet"'] \arrow[dr, "\varphi"] & \\ O_1^\bullet \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{40}$$

commutes and $\psi_1 E_1^\bullet = E'$. Let $\psi = \psi_1 + \{m^\bullet \mapsto m'\}$. Since (40) commutes and $m^\bullet$ is fresh, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

commutes. Also, since $\psi_1 E_1^\bullet = E'$ we have $\psi S^\bullet = (m', E')$, as required.

$\boxed{\text{Rule 5, } sigexp \equiv sigid}$

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash sigid \Rightarrow S'$. Let $(A, B) = O$ and $(A', B') = O'$. Then $sigid \in \mathrm{Dom}\, B'$ and $B'(sigid) \geq S'$. Then $sigid \in \mathrm{Dom}\, B$, so W does not fail here. As in W, write $B(sigid)$ in the form $(N^\bullet)S^\bullet$, where $N^\bullet \cap \mathrm{names}(O) = \emptyset$. Let $\varphi^\bullet = \mathrm{Id}$ and let $O^\bullet = (A^\bullet, B)$, where $A^\bullet = (S^\bullet, A)$. Then W succeeds with result $(O^\bullet, \varphi^\bullet, S^\bullet)$. Clearly, $O^\bullet$ is admissible and $O \xrightarrow{\varphi^\bullet} O^\bullet$, as required. Moreover, $(N^\bullet)S^\bullet \geq S^\bullet$ and $A^\bullet \sqsupseteq S^\bullet$, so $O^\bullet \vdash sigid \Rightarrow S^\bullet$.

Part 2: Let $O' = (A', B')$, φ and S' by any objects satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash sigid \Rightarrow S'$. Then $B'(sigid) \geq S'$, i.e., $\varphi((N^\bullet)S^\bullet) \geq S'$. Instead of trying to apply φ into $(N^\bullet)S^\bullet$ (which may involve renaming), it is easier to note that $\varphi((N^\bullet)S^\bullet) \geq S'$

holds precisely if there exists a realization ψ such that $\psi(S^\bullet) = S'$ and $\psi(m) = m$, for all m free in $(N^\bullet)S^\bullet$ (ψ does realization on free and bound names simultaneously). Indeed, since $N^\bullet \cap \text{names}(O) = \emptyset$, we can obtain $\psi(m) = \varphi(m)$, for all $m \in \text{names}(O)$. Thus the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

commutes and $\psi S^\bullet = S'$.

Rule 7, principal signatures

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow \Sigma'$. Since $O' \vdash sigexp \Rightarrow \Sigma'$ must be inferred by rule 7, Σ' must be principal for *sigexp* in O'. Thus, writing $\Sigma' = (N')S'$, where $N' \cap \text{names}(O') = \emptyset$, there exists A'_1 such that $A' \trianglelefteq A'_1$ and $A'_1, B' \vdash sigexp \Rightarrow S'$. Note that $N' \subseteq \text{names}(S')$, as Σ' is well-formed. We now wish to use the induction hypothesis to prove that *sigexp* elaborates to a principal signature $\Sigma^\bullet$ in some $O^\bullet$. To this end, let $O'_1 = (A'_1, B')$. Since $A' \trianglelefteq A'_1$ and $O \xrightarrow{\varphi} O'$ we have $O \xrightarrow{\varphi} O'_1$. Thus by the induction hypothesis on $A'_1, B' \vdash sigexp \Rightarrow S'$ and $O \xrightarrow{\varphi} O'_1$, the call $(O^\bullet \text{ as } (A^\bullet_1, B^\bullet), \varphi^\bullet, S^\bullet) = W_{sigexp}(O, sigexp)$ succeeds and $O \xrightarrow{\varphi^\bullet} O^\bullet_1$ and $O^\bullet_1 \vdash sigexp \Rightarrow S^\bullet$. Following W, let $A^\bullet = \text{Below}(A^\bullet_1, \varphi^\bullet(A))$, $O^\bullet = (A^\bullet, B^\bullet_1)$ and let $N^\bullet$ and $\Sigma^\bullet$ be given by $\Sigma^\bullet = (N^\bullet)S^\bullet = \text{Clos}_{A^\bullet} S^\bullet$. Then $W_{prinsigexp}(O, sigexp)$ returns $(O^\bullet, \varphi^\bullet, \Sigma^\bullet)$. By Lemma 6.11, $\Sigma^\bullet$ is principal for *sigexp* in $O^\bullet$. Since $O \xrightarrow{\varphi^\bullet} O^\bullet_1$ and $A^\bullet = \text{Below}(A^\bullet_1, \varphi^\bullet A)$ we have the desired $O \xrightarrow{\varphi^\bullet} O^\bullet$, by Lemma 6.2. Next we want to prove that $O^\bullet \vdash sigexp \Rightarrow \Sigma^\bullet$. This almost follows from the results of applying Lemma 6.11 above, but we have to prove that $A^\bullet \sqsupseteq \Sigma^\bullet$. Now $\Sigma^\bullet$ is well-formed (by Lemma 6.5 on $A^\bullet \trianglelefteq A^\bullet_1$ and $A^\bullet_1 \sqsupseteq S^\bullet$). But then $A^\bullet \sqsupseteq \Sigma^\bullet$ follows from the definition of $\Sigma^\bullet$ and the fact that $A^\bullet \trianglelefteq A^\bullet_1$ and $A^\bullet_1 \sqsupseteq S^\bullet$. Thus $O^\bullet \vdash sigexp \Rightarrow \Sigma^\bullet$ holds, as required.

Part 2: Now let $(O' \text{ as } (A', B'), \varphi, \Sigma')$ be such that $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow \Sigma'$. We then obtain A'_1, O'_1, N' and S' as above. By induction there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet_1 \arrow[rr, "\psi"] & & O'_1 \end{tikzcd} \tag{41}$$

commutes and $\psi S^\bullet = S'$. But then, by Lemma 6.10, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^\bullet"'] \arrow[dr, "\varphi"] & \\ O^\bullet \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{42}$$

commutes. By Lemma 6.7 we have $\Sigma' = \mathrm{Clos}_{A'}S'$. By Lemma 6.13 we then get

$$\psi((N^*)S^*) = (\psi N^*)(\psi S^*) = (N')(\psi S^*)$$

Since $\psi S^* = S'$ we thereby have $\psi\Sigma^* = \Sigma'$, as required.

Rule 8, *funsigexp* $\equiv$ (*strid* : *sigexp*$_1$) : *sigexp*$_2$

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash funsigexp \Rightarrow \Phi'$. Then by rule 8, Φ' is of the form $(N_1')(S_1', \Sigma_2')$ and $O' \vdash sigexp_1 \Rightarrow (N_1')S_1'$ and

$$(S_1', A'), B' + N_1' + \{strid \mapsto S_1'\} \vdash sigexp_2 \Rightarrow \Sigma_2' \tag{43}$$

where $(A', B') = O'$ and $N_1' \cap \mathrm{names}(O') = \emptyset$.

By induction on $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp_1 \Rightarrow (N_1')S_1'$, the call ($O_1^*$ as (A_1^*, B_1^*), φ_1^*, Σ_1^* as $(N_1^*)S_1^*) = W_{prinsigexp}(O, sigexp_1)$ succeeds and $O \xrightarrow{\varphi_1^*} O_1^*$ and

$$O_1^* \vdash sigexp_1 \Rightarrow (N_1^*)S_1^* \tag{44}$$

Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{45}$$

commutes and $\psi_1((N_1^*)S_1^*) = (N_1')S_1'$.

Without loss of generality, we may assume that $N_1^* \cap \mathrm{names}(O_1^*, O') = \emptyset$; in addition we can assume that $N_1^* = N_1'$ and that ψ_1 is fixed on N_1^*. Notice that with these assumptions, $\psi_1(N_1^*) = N_1'$, $\psi_1 S_1^* = S_1'$ and

$$\psi_1((N_1^*)S_1^*) = (\psi_1 N_1^*)\ (\psi_1 S_1^*) = (N_1')S_1' \tag{46}$$

Let

$$\begin{array}{ll} A_2 = (S_1^*, A_1^*) & A_2' = (S_1', A') \\ B_2 = B_1^* + N_1^* + \{strid \mapsto S_1^*\} & B_2' = B' + N_1' + \{strid \mapsto S_1'\} \\ O_2 = (A_2, B_2) & O_2' = (A_2', B_2') \end{array}$$

We have $O_2 \xrightarrow{\psi_1} O_2'$. (To see this, first note that O_2' is an object in K by (43); similarly, O_2 is an object in K by (44). Moreover, $\psi_1 A_2 \sqsubseteq A_2'$, since $\psi_1 A_1^* \sqsubseteq A'$ and $\psi_1 S_1^* = S_1'$. Finally, A_2 covers A_2' on N of $B_2 = (N$ of $B) \cup N_1^*$, for A_1^* covers A' on N of B so by (46) and the fact that $A_1^* \sqsupseteq (N_1^*)S_1^*$ and $A' \sqsupseteq (N_1')S_1'$ we have that A_2 covers (S_1', A') on $(N$ of $B) \cup N_1^*$.) Also, by (43) we have $O_2' \vdash sigexp_2 \Rightarrow \Sigma_2'$. By induction on $O_2 \xrightarrow{\psi_1} O_2'$ and $O_2' \vdash sigexp_2 \Rightarrow \Sigma_2'$, the call $(O_2^*, \varphi_2^*, \Sigma_2^*) = W_{prinsigexp}(O_2, sigexp_2)$ succeeds and $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and

$$O_2^* \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{47}$$

Moreover, there exists a ψ_2 such that the diagram

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O_2^* \arrow[rr, "\psi_2"] & & O_2' \end{tikzcd} \tag{48}$$

commutes and $\psi_2\Sigma_2^* = \Sigma_2'$. Let $(A_2^*, B_2^*) = O_2^*$. By the definition of $W_{prinsigexp}$ we know that $A_2^* = \text{Below}(A_0, \varphi_2^*(A_2))$, for some A_0. Thus $A_2^* \sim \text{Below}(A_2^*, \varphi_2^*(A_2))$. Following the definition of W, let $A^* = \text{Below}(A_2^*, \varphi_2^*(A_1^*))$. By Lemma 6.14 we have

$$A_2^* \sim (\varphi_2^* S_1^*, A^*) \tag{49}$$

$$N_1^* \cap \text{names}(A^*) = \emptyset \tag{50}$$

and also that the diagram

$$\begin{tikzcd} & O_1^* \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O^* \arrow[rr, "\psi_2"] & & O' \end{tikzcd} \tag{51}$$

commutes, where $O^* = (A^*, \varphi_2^*(B_1^*))$. Thus, by (50), $W(O, funsigexp)$ does not fail here; on the contrary, it succeeds with result (O^*, φ^*, Φ^*), where $\varphi^* = \varphi_2^* \circ \varphi_1^*$ and $\Phi^* = (N_1^*)(\varphi_2^* S_1^*, \Sigma_2^*)$.

By composing the diagrams (45) and (51) we get the desired $O \xrightarrow{\varphi^*} O^*$. By Theorem 5.1 on (44) we get

$$O^* \vdash sigexp_1 \Rightarrow \varphi_2^*((N_1^*)S_1^*) \tag{52}$$

Also, since φ_2^* is fixed on N_1^* and $\Sigma_1^* \sqsubseteq A_1^*$ and $\varphi_2^* A_1^* \sqsubseteq A^*$ and (50) we have $\varphi_2^*((N_1^*)S_1^*) = (N_1^*)(\varphi_2^* S_1^*)$. Thus (52) reads $O^* \vdash sigexp_1 \Rightarrow (N_1^*)\varphi_2^* S_1^*$, i.e.

$$A^*, \varphi_2^* B_1^* \vdash sigexp_1 \Rightarrow (N_1^*)\varphi_2^* S_1^* \tag{53}$$

Expanding (47) we get

$$A_2^*, \varphi_2^* B_1^* + N_1^* + \{strid \mapsto \varphi_2^* S_1^*\} \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{54}$$

Thus by Lemma 3.1 on (54) and (49) we have

$$(\varphi_2^* S_1^*, A^*), \varphi_2^* B_1^* + N_1^* + \{strid \mapsto \varphi_2^* S_1^*\} \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{55}$$

We now wish to use rule 8 on (53) and (55). The side-condition $N_1^* \cap \text{names}(A^*) = \emptyset$ is met by (50). Moreover, Φ^* is well-formed, for the following reasons. $(N_1^*)(\varphi_2^* S_1^*)$ is well-formed by (53), Σ_2^* is well-formed by (54) and if (m, E) is a structure occurring in Σ_2^* with $m \notin N_1^*$ then since $A_2^* \sqsupseteq \Sigma_2^*$ and (49) we have $m \in \text{names}(A^*)$. Since $A^* \trianglelefteq A_2^*$ we therefore have $(m, \text{skel}(E)) \sqsubseteq A^*$, so $\text{names}(\text{skel}(E)) \cap N_1^* = \emptyset$, showing that Φ^* is well-formed and that $A^* \sqsupseteq \Phi^*$. Thus rule 8 applies and we get the desired $O^* \vdash funsigexp \Rightarrow \Phi^*$.

Part 2: Let O', φ and Φ' be objects satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash funsigexp \Rightarrow \Phi'$. By using induction twice, we get diagrams (45), (48) and (51) exactly as above, but with ψ_1 and ψ_2 depending on the new choice of O', φ and Φ'. Let $\psi = \psi_2$. We wish to prove that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{56}$$

commutes and that $\psi\Phi^* = \Phi'$. But (56) commutes since (45) and (51) commute. Moreover, as $A^* \sqsupseteq \Phi^*$ and $O^* \xrightarrow{\psi} O'$ and $\mathrm{names}(O^*) \cap N_1^* = \mathrm{names}(O') \cap N_1^* = \emptyset$, we can perform the application $\psi(\Phi^*) = \psi((N_1^*)(\varphi_2^* S_1^*, \Sigma_2^*)) = (N_1^*)(\psi\varphi_2^* S_1^*, \psi\Sigma_2^*)$ directly, without causing name capture. Thus

$$\begin{array}{rcll} \psi(\Phi^*) & = & (N_1^*)(\psi\varphi_2^* S_1^*, \psi\Sigma_2^*) & \\ & = & (N_1^*)(\psi_1 S_1^*, \Sigma_2') & \text{as (48) commutes} \\ & = & (N_1')(S_1', \Sigma_2') & \text{by (46)} \\ & = & \Phi' & \end{array}$$

as required.

Rule 15, *shareq* $\equiv$ *longstrid*$_1$ = *longstrid*$_2$

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{longstrid}_1 = \mathit{longstrid}_2 \Rightarrow \{\}$. Let $(A, B) = O$ and $(A', B') = O'$. By the side-condition on rule 15 we have m of $(B'(\mathit{longstrid}_1)) = m$ of $(B'(\mathit{longstrid}_2))$, but the problem is that we do not necessarily have m of $(B(\mathit{longstrid}_1)) = m$ of $(B(\mathit{longstrid}_2))$. (One reason is that whenever we choose names in the proof, e.g. in the cases for rules 4 and 5, we always choose them to be suitably 'new'.) Since $O \xrightarrow{\varphi} O'$ and $B'(\mathit{longstrid}_1)$ and $B'(\mathit{longstrid}_2)$ exist and have the same name, W does not fail when it computes $(m_1, E_1) = B(\mathit{longstrid}_1)$ and $(m_2, E_2) = B(\mathit{longstrid}_2)$. Also, φ is a unifier for m_1 and m_2 in A under N of B. By Theorem 4.1, the call $\varphi^* = \mathit{Unify}(A, N \text{ of } B, (m_1, m_2))$ succeeds and φ^* is a most general unifier for m_1 and m_2 in A under N of B. Thus $W(O, \mathit{shareq})$ succeeds with result (O^*, φ^*, E^*), where $O^* = \varphi^*(O)$ and $E^* = \{\}$. We have $O \xrightarrow{\varphi^*} O^*$ and $O^* \vdash \mathit{shareq} \Rightarrow E^*$, since φ^* is a unifier for m_1 and m_2 in A under N of B.

Part 2: Let O', φ and E' be arbitrary objects satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{shareq} \Rightarrow E'$. Then φ is a unifier for m_1 and m_2 in A under N of B. Since φ^* is a most general such unifier, there exists a ψ which is fixed on N of B and satisifies $\psi(\varphi^*(A)) = \varphi(A)$. Thus the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd}$$

commutes and $\psi E^* = E'$.

Rule 14, *spec* $\equiv$ *atspec*$_1$ $\langle ; \rangle$ *spec*$_2$

Part 1: Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{atspec}_1 \langle ; \rangle \mathit{spec}_2 \Rightarrow E'$. Let $(A', B') = O'$. By

rule 14 there exist E'_1 and E'_2 such that $E' = E'_1 + E'_2$ and $A', B' \vdash atspec_1 \Rightarrow E'_1$ and $A', B' + E'_1 \vdash spec_2 \Rightarrow E'_2$. By induction on $O \xrightarrow{\varphi} O'$ and $O' \vdash atspec_1 \Rightarrow E_1$, we have that the call $(O_1^* \text{ as } (A_1^*, B_1^*), \varphi_1^*, E_1^*) = W_{atspec}(O, atspec_1)$ succeeds and $O \xrightarrow{\varphi_1^*} O_1^*$ and $O_1^* \vdash atspec_1 \Rightarrow E_1^*$. Moreover, there exists a ψ_1 such that the diagram

$$
\begin{tikzcd}
 & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\
O_1^* \arrow[rr, "\psi_1"] & & O'
\end{tikzcd}
\tag{57}
$$

commutes and $\psi_1 E_1^* = E'_1$. Let $O_2 = (A_1^*, B_1^* + E_1^*)$ and $O'_2 = (A', B' + E'_1)$. Then $O_2 \xrightarrow{\psi_1} O'_2$ and $O'_2 \vdash spec_2 \Rightarrow E'_2$. By induction, the call $(O_2^*, \varphi_2^*, E_2^*) = W_{spec}(O_2, spec_2)$ succeeds and $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and $O_2^* \vdash spec_2 \Rightarrow E_2^*$. Moreover, there exists a ψ_2 such that

$$
\begin{tikzcd}
 & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\
O_2^* \arrow[rr, "\psi_2"] & & O'_2
\end{tikzcd}
\tag{58}
$$

commutes and $\psi_2 E_2^* = E'_2$. Let $(A_2^*, B_2^*) = O_2^*$, $O^* = (A_2^*, \varphi_2^*(B_1^*))$, $\varphi^* = \varphi_2^* \circ \varphi_1^*$ and $E^* = \varphi_2^* E_1^* + E_2^*$. Then $W_{spec}(O, spec)$ succeeds with result (O^*, φ^*, E^*). We wish to prove $O \xrightarrow{\varphi^*} O^*$ and

$$O^* \vdash atspec_1 \;\langle ; \rangle\; spec_2 \Rightarrow E^* \tag{59}$$

Since $B_1^* \sqsubseteq A_1^*$ we have $\varphi_2^* B_1^* \sqsubseteq \varphi_2^* A_1^* \sqsubseteq A_2^*$. Thus O^* is an object in K and $O_1^* \xrightarrow{\varphi_2^*} O^*$ holds. Since (58) commutes we then have that

$$
\begin{tikzcd}
 & O_1^* \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\
O^* \arrow[rr, "\psi_2"] & & O'
\end{tikzcd}
\tag{60}
$$

commutes. We get $O \xrightarrow{\varphi^*} O^*$ by composition of (57) and (60). Moreover, by Theorem 5.1 on $O_1^* \vdash atspec_1 \Rightarrow E_1^*$ and $O_1^* \xrightarrow{\varphi_2^*} O^*$ we have $O^* \vdash atspec_1 \Rightarrow \varphi_2^* E_1^*$. Also $O_2^* = (A_2^*, B_2^*) = (A_2^*, \varphi_2^* B_1^* + \varphi_2^* E_1^*)$, so from $O_2^* \vdash spec_2 \Rightarrow E_2^*$ we get $O^* + \varphi_2^* E_1^* \vdash spec_2 \Rightarrow E_2^*$. Thus by rule 14 we have (59) as required.

Part 2: Now let O', φ and E' be arbitrary objects satisfying $O \xrightarrow{\varphi} O'$ and $O' \vdash spec \Rightarrow E'$. By proceeding as above, we obtain ψ_1 and ψ_2 (and O_2, O'_2) depending on the new O', φ and E' such that the diagrams (57), (58) and (60) commute. Let $\psi = \psi_2$. Since (57) and (60) commute, the diagram

$$
\begin{tikzcd}
 & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\
O^* \arrow[rr, "\psi"] & & O'
\end{tikzcd}
$$

commutes and moreover, since (58) commutes, we have $\psi E^{\bullet} = \psi(\varphi_2^{\bullet}E_1^{\bullet} + E_2^{\bullet}) = \psi\varphi_2^{\bullet}E_1^{\bullet} + \psi E_2^{\bullet} = \psi_1 E_1^{\bullet} + \psi E_2^{\bullet} = E_1' + E_2' = E'$, as required.

7 Conclusion

We have presented a language, called HML, which is intended for programming with higher-order modules. From a programmer's point of view, HML is largely compatible with Standard ML, but HML allows functors to be declared in structures and specified in signatures.

We have defined the static semantics of signature expressions and shown that if a signature expression elaborates at all, then it can be elaborated to a principal signature, using an algorithm, which we have presented and proved correct. We have not defined the semantics of structure matching or functor application.

Part of the motivation behind the work reported in this paper was to see whether, and how smoothly, the semantics of first-order functors presented in the Definition of Standard ML could be extended to higher-order functors. As far as the semantic objects are concerned, a major difference is that one now has to deal with nested quantification. Also, in some situations, it is important to distinguish between the substructures of a given structure S and those structures that only occur (free) inside some functor signature inside S.

In HML we have to be able to find principal signatures in bases that contain flexible structures. This is not the case in Standard ML. Also, for type-checking reasons, we need principality to be preserved under realization, which is not the case in Standard ML. This has been achieved first and foremost by introducing assemblies into the inference rules and by promoting the notion of cover to ensure that the assembly serves as a consistent frame of reference for different views of structures. These changes have had effects that are useful, even for first-order modules. For example, principal signatures are now always well-formed.

With one exception, we feel that the semantics of Standard ML signature expressions scaled well to the higher-order language. The exception is local and overlapping sequential specifications, which have rather subtle implications for the semantics (see the discussions in sections 3.3 and 3.5).

The algorithms in this paper have been implemented in the ML Kit, by Lars Birkedal, who has also extended the theorems and proofs presented in this paper to cover all the constructs found in Standard ML signatures. David MacQueen and Pierre Cregut have recently implemented higher-order functors, including functor application, in Standard ML of New Jersey.

Acknowledgements

This work would have been impossible without previous work with Robin Milner and Robert Harper on the semantics of Standard ML. Also, I want to thank Maria-Virginia Aponte, Lars Birkedal, David MacQueen and David N. Turner for valuable discussions. Special thanks to the referees for their very detailed and constructive suggestions and comments.

This work is supported by grant number 5.21.08.03 from the Danish Research Council as part of the DART project.

Appendix: Proof of Lemma 6.14

Let $(A_2, B_2) = O_2$ and $(A'_2, B'_2) = O'_2$. We first prove

$$A_2^* \sim (A^*, \varphi_2^*(A_2)) \tag{61}$$

We have $(A^*, \varphi_2^* A_2) \sqsubseteq A_2^*$ by the definition of A^* and the fact that $O_2 \xrightarrow{\varphi_2^*} O_2^*$. Let us show the converse $A_2^* \sqsubseteq (A^*, \varphi_2^* A_2)$. Since by assumption $A_2^* \sqsubseteq \mathrm{Below}(A_2^*, \varphi_2^* A_2)$ it will suffice to prove

$$\mathrm{Below}(A_2^*, \varphi_2^* A_2) \sqsubseteq (A^*, \varphi_2^* A_2) \tag{62}$$

Consider the sequence $N_0 \subseteq N_1 \subseteq N_2 \subseteq \cdots$ of name sets defined by letting $N_0 = \emptyset$ and, for $i \geq 0$,

$$\begin{aligned} N_{i+1} \quad = \quad & \{m \in \mathrm{names}(A_2) \mid \\ & \forall E.\text{if } (m, E) \text{ occurs free in } A_2, \text{ then } \mathrm{names}(\mathrm{skel}(E)) \subseteq N_i\} \end{aligned}$$

Note that $N_{k+1} = N_k = \mathrm{names}(A_2)$, for some k, since A_2 is finite and cycle-free. Thus we can prove (62) by proving by induction on i that $\mathrm{Below}(A_2^*, \varphi_2^* N_i) \sqsubseteq (A^*, \varphi_2^* A_2)$.
Base Case, $i = 0$. Here $\mathrm{Below}(A_2^*, \emptyset)$ is the empty assembly which trivially is covered by $(A^*, \varphi_2^* A_2)$.
Inductive Step, $i > 0$. Assume $\mathrm{Below}(A_2^*, \varphi_2^* N_{i-1}) \sqsubseteq (A^*, \varphi_2^* A_2)$. We wish to prove that $\mathrm{Below}(A_2^*, \varphi_2^* N_i) \sqsubseteq (A^*, \varphi_2^* A_2)$. As we already have $\mathrm{Below}(A_2^*, \varphi_2^* N_{i-1}) \sqsubseteq (A^*, \varphi_2^* A_2)$, it will suffice to prove that for all (m, E) occurring free in A_2^* with $m \in \mathrm{below}(A_2^*, \varphi_2^* N_i) \setminus \mathrm{below}(A_2^*, \varphi_2^* N_{i-1})$ that $(m, \mathrm{skel}(E)) \sqsubseteq (A^*, \varphi_2^* A_2)$. So let (m, E) be such a structure. As $m \in \mathrm{below}(A_2^*, \varphi_2^* N_i)$ there exists an $m_i \in N_i \setminus N_{i-1}$ such that $m \in \mathrm{below}(A_2^*, \varphi_2^* m_i)$. There are two cases to consider:
Case 1, $m_i \in N_1^*$. As φ_2^* is fixed on N_1^*, we have $\varphi_2^* m_i = m_i$ and hence $m \in \mathrm{below}(A_2^*, m_i)$. However, since $m \notin \mathrm{below}(A_2^*, \varphi_2^* N_{i-1})$ and A_2 covers A_2^* on N_1^*, we must have $m = m_i$.
Let *strid* be a structure identifier in $\mathrm{Dom}\, E$. Since A_2 covers A_2^* on N_1^* there exists an environment E_1 such that (m, E_1) occurs free in A_2 and $strid \in \mathrm{Dom}\, E_1$. Thus $\varphi_2^*(m, E_1) \sqsubseteq \varphi_2^* A_2$. But then $\varphi_2^*(m \text{ of } E_1(strid)) = m \text{ of } E(strid)$, since A_2^* is consistent. Also, $m \text{ of } (E_1(strid)) \subseteq N_{i-1}$, so $\mathrm{skel}(E(strid)) \sqsubseteq \mathrm{Below}(A_2^*, \varphi_2^* N_{i-1}) \sqsubseteq (A^*, \varphi_2^* A_2)$, by the induction hypothesis. Thus

$$(m, \mathrm{skel}(SE \text{ of } E)) \sqsubseteq (A^*, \varphi_2^* A_2) \tag{63}$$

Next, let *funid* be a functor identifier in $\mathrm{Dom}\, E$. Since A_2 covers A_2^* on N_1^* there exists an environment E_1 such that (m, E_1) occurs free in A_2 and $funid \in \mathrm{Dom}\, E_1$ and $(m, \varphi_2^* E_1) = \varphi_2^*(m, E_1) \sqsubseteq \varphi_2^* A_2$. Thus

$$(m, \mathrm{skel}(FE \text{ of } E)) \sqsubseteq \varphi_2^* A_2 \tag{64}$$

But from (63) and (64) we immediately get $(m, \mathrm{skel}(E)) \sqsubseteq (A^*, \varphi_2^* A_2)$, as required.

Case 2, $m_i \in \text{names}(A_1^\bullet)$. Since $m \in \text{below}(A_2^\bullet, \varphi_2^\bullet m_i)$ we then have $(m, \text{skel}(E)) \sqsubseteq A^\bullet$, by the definition of $A^\bullet$. This proves (61).

But then $A_2^\bullet \sim (\varphi_2^\bullet S_1^\bullet, A^\bullet)$ for the following reason. Since $A^\bullet = \text{Below}(A_2^\bullet, \varphi_2^\bullet A_1^\bullet)$ and $O_2 \xrightarrow{\varphi_2^\bullet} O_2^\bullet$ we have $\varphi_2^\bullet A_1^\bullet \sqsubseteq A^\bullet$. Thus $\varphi_2^\bullet A_2 = (\varphi_2^\bullet S_1^\bullet, \varphi_2^\bullet A_1^\bullet) \sqsubseteq (\varphi_2^\bullet S_1^\bullet, A^\bullet)$. Thus $(\varphi_2^\bullet S_1^\bullet, A^\bullet) \sim (A^\bullet, \varphi_2^\bullet A_2) \sim A_2^\bullet$. This proves the desired $(\varphi_2^\bullet S_1^\bullet, A^\bullet) \sim A_2^\bullet$.

That $N_1^\bullet \cap \text{names}(A^\bullet) = \emptyset$ is seen as follows. Assume $N_1^\bullet \cap \text{names}(A^\bullet) \neq \emptyset$, and let m be an element of $N_1^\bullet \cap \text{names}(A^\bullet)$. Since $m \in \text{names}(A^\bullet)$, there exists an $m' \in \text{names}(A_1^\bullet)$ such that $m \in \text{below}(A_2^\bullet, \varphi_2^\bullet m')$. Then, since $O_2^\bullet \xrightarrow{\psi} O_2'$ we have $\psi m \in \text{below}(A_2', \psi(\varphi_2^\bullet m'))$, i.e. $m \in \text{below}(A_2', \psi_1 m')$, since diagram (b) of Fig. 23 commutes and all the realizations are fixed on $N_1^\bullet$. Since $m' \notin N_1^\bullet$ and $\psi_1 A_1^\bullet \sqsubseteq A'$ we have $\psi_1 m' \in \text{names}(A')$. Since $A' \trianglelefteq A_2'$ we therefore have $\text{below}(A_2', \psi_1 m') \subseteq \text{names}(A')$, so $m \in \text{names}(A')$. But $m \in \text{names}(A') \cap N_1^\bullet$ contradicts the assumption $N_1^\bullet = \text{names}(A_2') \setminus \text{names}(A')$.

Finally, let us show that Fig. 23(c) exists and commutes. Since $A_1^\bullet \trianglelefteq A_2$ and $A^\bullet = \text{Below}(A_2^\bullet, \varphi_2^\bullet A_1^\bullet)$ and $O_2 \xrightarrow{\varphi_2^\bullet} O_2^\bullet$ we have $O_1^\bullet \xrightarrow{\varphi_2^\bullet} O^\bullet$, by Lemma 6.2. This shows that the left-hand morphism exists. The right-hand morphism exists by assumption. But then, since (b) commutes, the bottom morphism exists and (c) commutes; the fact that $O^\bullet$ covers O' on N of B follows from Lemma 6.9. □

References

Aït–Kaci, H. (1986) An algebraic semantics approach to the effective resolution of type equations. *Theoretical Comput. Sci.*, 45(3): 293–351.

Aponte, M. V. (1992) Typage d'un système de modules paramétriques avec partage: une application de l'unification dans les théories équationnelles. PhD thesis, INRIA, University of Paris VII, France.

Aponte, M. V. (1993) Extending record typing to type parametric modules with sharing. In: *Proc. 20th ACM Symp. on Principles of Program. Lang. (POPL)*, pp. 465–478.

Damas, L. and Milner, R. (1982) Principal type schemes for functional programs. In: *Proc. 9th Ann. ACM Symp. on Principles of Program. Lang. (POPL)*, pp. 207–212.

Harper, R., Mitchell, J. and Moggi, E. (1990) Higher-order modules and the phase distinction. In: *Proc. ACM Symp. on Principles of Program. Lang. (POPL)*, pp. 341–354.

Harper, R., Milner, R. and Tofte, M. (1987) A type discipline for program modules. In: *Proc. Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Lecture Notes in Computer Science 250*, Springer, pp. 308–319.

MacQueen, D. B. (1984) Modules for Standard ML. In: *Conf. Record ACM Symp. on LISP and Functional Program.*, pp. 198–207.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Computer and System Sci.*, **17**: 348–375.

Milner, R. and Tofte, M. (1991) *Commentary on Standard ML*, MIT Press.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*, MIT Press.

Rémy, D. (1989) Typechecking records and variants in a natural extension of ML. In: *Proc. 16th Ann. ACM Symp. on Principles of Program. Lang. (POPL)*, pp. 77–88.

Tofte, M. (1988) Operational semantics and polymorphic type inference. PhD thesis, Department of Computer Science, Edinburgh University. (Also available as Technical Report CST-52-88.)

Wand, M. (1989) Type inference for record concatenation and multiple inheritance. In: *Proc. IEEE 4th Ann. Symp. on Logic in Comput. Sci. (LICS)*, IEEE Press, pp. 92–97.