Higher-order Functors and Principal Signatures in Standard ML

Lars Birkedal
DIKU, Department of Computer Science
University of Copenhagen
DK–2100 Copenhagen Ø
Denmark
Email: `birkedal@diku.dk`

September 1993

Contents

Abstract

In this report we present an extension of the Standard ML Modules Language [12], where specification of functors inside signatures and declaration of functors inside structures is allowed, thereby making it possible to parameterise modules on parameterised modules; in effect providing higher-order modules. (The Standard ML Modules Language is only first-order.) We show that principal signatures still exist, and give an algorithm which computes the principal signature for a signature expression if one exists. Moreover, we describe how illegal signature expressions are detected. The algorithm, including detection of illegal signature expresssions, has been implemented in the ML Kit [3] — we provide a description of the implementation. In this report we are only concerned with static semantics for signature expressions and signature bindings.

Preface

This report is submitted in partial fulfillment of of the requirements for a Danish Master's Degree (Candidatus Scientiarum). It contains work done under the supervision of Mads Tofte at DIKU, the Department of Computer Science at the University of Copenhagen. The work reported here is an extension of Tofte's work on a higher-order skeletal language, called HML [15] [16].

Outline

The report is organized as follows. In Chapter 2 we briefly define the extension of the syntax of SML. In the next chapter we discuss how to extend the static semantics of SML Modules to higher-order modules. This leads to the definition of a static semantics for the higher-order modules language; the semantics is presented in Chapter 4. In Chapter 5 we state and prove some propositions concerning the elaboration of signature expressions, and in Chapter 6 we state and prove the existence of principal signatures. Chapter 7 presents an algorithm for inferring principal signatures and describes how to detect illegal signature expressions. In Chapter 8 we give an overview of the implementation of the algorithm in the ML Kit, and finally we conclude in Chapter 9.

Reader's Prerequisites

We assume the reader is familiar with the Definition of Standard ML [12] and the Commentary on Standard ML [11]. Henceforth [12] will be referred to simply as "the Definition" and [11] as "the Commentary". Throughout this paper, we use the notation and terminology of the Definition and the Commentary.

As we do not assume that the reader is familiar with Tofte's work on HML, we will repeat some of his work. In most of the proofs we have either needed to extend the proofs because we treat the full language or have written some more details. I have, however, painstakingly tried to remark where each piece is taken from and whether or not I have added anything.

Acknowledgements

Many thanks to Mads Tofte for his splended supervision of this work; he has been a very inspiring supervisor. Moreover, it was he who introduced me to Standard ML and polymorphic type disciplines and suggested this project. Also thanks to Mads Tofte for employing me as ML Kit programmer — it has been, and certainly still is, great fun working for and with him. Thanks to Lars Thorup for giving constructive comments on parts of this paper.

Chapter 1

Introduction

The Modules Language of Standard ML supports working on large programs. The central concepts of the Modules language are *structures*, *signatures* and *functors* [8][9][12]. A structure can *declare* datatypes and functions operating on these types. A signature can *specify* names and types of functions; it can be thought of as the "type" of a structure. A functor is a parameterised module; it can be thought of as a function from structures (matching a certain signature corresponding to a function argument beeing of a certain type) to structures.

One of the practical advantages of programming with functors is that it is possible to write a functor F , and have it elaborated[1] before the actual argument to which F will be applied is written. We would also like to be able to write a functor F_1 which depends upon another functor F_2 and have F_1 elaborated before F_2 is written; this is not possible in general in the SML Modules language as the following example shows. I would like to stress that the example is not a constructed one — the author ran into it while programming a real piece of software.

Example 1 Suppose we want to program a well modularised elaborator for a little applicative polymorphic language, with integer and boolean constants, functions and pairs. We choose to use an algorithm resembling algorithm W [10] [4]. Unification of two types should not return a substitution, instead unifying one type with another should simply, in some way or another, destructively update the type variables with the types they are unified with. The idea is that we will avoid having to apply a substitution to a type environment, hoping for better efficiency. Our elaborator will operate on static objects

[1] We use the term *elaboration* for "the static part of execution" instead of the often used term *type inference* because other semantic objects than types are inferred.

like types and type enviroments and it must be possible to unify two types. Types are first-order terms, and we therefore choose to represent types by a generic implementation of first-order terms parameterised on the set of term constructors. This leads us to the following signatures for terms and static objects.

```
signature TERM =
  sig
    type constructor
     and term
    val mk_var : unit -> term
    and mk_con : constructor * term list -> term
    ...
    val unify  : ...
  end;
signature STATOBJECT =
  sig
    datatype TypeConstructors = INT | BOOL | ARROW | PAIR
    structure Term : TERM
    sharing type Term.constructor = TypeConstructors
    type Type
     and TypeEnvironment
     and var
    ...
    val lookupTE : var * TypeEnvironment -> ...
    ...
    exception Unify
    val unify : Type * Type -> Type
  end;
```

Notice that there are good reasons for parameterising this much; an implementation of first-order terms can be used in other situations and it makes it possible for us to make two different implementations of first-order terms, e.g. terms represented by term graphs [6] (where the equivalence class representative of a variable node is changed when the variable is unified with another term) and terms where the variables are implemented as a reference to a term (such that the variable is destructively updated when it is unified with another term), and compare the practical efficiency of the two implementations when elaborating real programs.

Having written the signatures we continue by implementing functors returning structures matching the signatures. This can be done as suggested by the following

```
functor Term(X: sig eqtype constructor end) :
   sig
     include TERM
     sharing type X.constructor = constructor
   end = struct ... end

functor StatObject() : STATOBJECT =
  struct
    type constructor = INT | BOOL | ARROW | PAIR
    structure Term = Term(constructor)
    ...
   end
```

However, it is not possible to write functor `StatObject` and have it elaborated before we have written functor `Term`. We would like to be able to parameterise functor `StatObject` on functor `Term` in the following way

```
functor StatObject(
          functor Term(X : sig eqtype constructor end):
            sig include TERM
                sharing type X.constructor = constructor
            end): STATOBJECT =
  struct
    type constructor = INT | BOOL | ARROW | PAIR
    structure Term = Term(constructor)
    ...
   end
```

such that `StatObject` could be elaborated before functor `Term` is written.

We should mention that there are other solutions to the above problem; instead of parameterising `Term` on the type "constructor", one could have specified the type `term` in signature `TERM` as a polymorphic type `''constructor term`. That would also make it possible to write functor `StatObject` before functor `Term`. However, that solution is not preferable, because it would not be as legible. □

Consider the elaboration of functors [11, Section 8] in Standard ML. Above we have used the derived form for functors; it suffices to consider functors of the simple form

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

as the functor arguments can always be wrapped up in a single structure (we use $\langle\rangle$ to enclose optional phrases). When the body of the functor is elaborated, the *strid* is bound to a formal structure S (think of it as a type) which has all the sharing, the polymorphism, and the components that any structure that matches the signature must have, but no more. Such an S exists because the *sigexp* can be elaborated to a unique principal signature $(N')S'$ (think of a signature as a type scheme [4]); we can let S be S' (the most general instance of the signature). The existence of principal signatures is one of the most, if not the most, important properties of the SML Modules Language. It has substantial practical significance; if no principal signature existed one would have to re-elaborate the functor body for each functor application.[2]

In this report we consider the problem of existence and inference of principal signatures for an extension of Standard ML's signature expressions in which it is possible to specify functors inside signatures. We define an extension of Standard ML, called HOF, largely compatible with SML, and define the static semantics of HOF signature expressions and signature bindings. Moreover, we prove that if a signature expression elaborates to a signature, then it elaborates to a principal signature, and we develop an algorithm which computes the principal signature if one exists, and otherwise fails. Our work is an extension of Tofte's work [15] [16]. Tofte defines a higher-order skeletal modules language, called HML. HML is a pure modules language in which specification of types, datatypes, values, and exceptions is not possible. Tofte defines the static semantics of signature expressions and proves that if a signature expression elaborates at all, then it can be elaborated to a principal signature, using an algorithm which he presents. Also, of course, our work builds upon the work on Standard ML [12] [11]. In the authors view, it is not trivial to extend Tofte's work; we have to consider problems with types and the

[2]As remarked in [11] one could also record a possibly infinite set of structures to which the *strid* could be bound, but this would in practice amount to the same: repeated elaboration of the body of the functor.

amount of details is increased considerably. The most difficult part has been to prove the existence of principal elaborations. Moreover, the problem in itself is relatively difficult as there is nested quantification in the higher-order signatures, and yet we are able to obtain a principality theorem.

Chapter 2

Syntax

HOF is Standard ML extended with the possibility of specifying functors in signatures and declaring functors inside structures.

2.1 Grammar

The grammar of Standard ML is extended by adding the productions in Figure 2.1 to the grammar rules in Figures 6–8 in the Definition. We have included the grammar rule for declaring functors inside structures to have the syntax settled for future examples, although we in this paper only will be concerned with signature bindings and signature expressions.

We have the same syntactic restrictions for HOF as for SML [12, Section 3.5] [11, Appendix D, Page 147].

spec	::=	`functor` *fundesc*	functor
fundesc	::=	*funid funsigexp* ⟨`and` *fundesc*⟩	
strdec	::=	`functor` *funbind*	

Figure 2.1: Grammar

Chapter 3

Discussion of Static Semantics

In this chapter we discuss informally how one can extend the semantics of modules to higher-order modules.

First, however, we recall the crucial distinction between rigid and flexible structures in the Standard ML Modules Language.

3.1 Rigid versus Flexible Structures

Consider the following piece of Standard ML code (taken from the Commentary)

```
structure Empty = struct end;

signature NONEMPTY =
sig
  structure NonEmpty :
    sig
      structure Dangle : sig end
    end
  sharing NonEmpty = Empty
end;
```

The signature expression for `NONEMPTY` does not elaborate in SML. Formally, is is because the signature for `NONEMPTY` is not covered by the basis and because it is not well-formed. Informally, the reason is that `NONEMPTY` can never be matched by any real structure, since that real structure should have a substructure `NonEmpty` which should share with `Empty`, and it is not possible to obtain a structure sharing with `Empty` which has a component `Dangle` — as the only way one can obtain a structure which shares with `Empty` (a *view* of `Empty`) is by cutting down `Empty` by a signature constraint, and empty does not have a substructure `Dangle`.

Now consider the following piece of SML code.

```
signature SIG =
sig
  structure Empty' : sig end
  structure NonEmpty' :
    sig
      structure NotDangle : sig end
    end
  sharing NonEmpty' = Empty'
end;
```

The signature `SIG` specifies two structures which must be views of the same structure (because of the sharing constraint.) The signature elaborates in SML as one would expect. Let us try to elaborate the signature expression for `SIG` in a left-to-right manner. `Empty'` elaborates to the structure $(m_1, \{\})$ and `NonEmpty'` elaborates to the structure $(m_2, \{\texttt{NotDangle}' \mapsto (m_3, \{\})\})$. Due to the sharing constraint m_1 and m_2 will be unified such that the outermost signature expression elaborates to the following structure

$$S = (m_0, \{\texttt{Empty}' \mapsto (m_1, \{\}), \texttt{NonEmpty}' \mapsto (m_1, \{\texttt{NotDangle} \mapsto (m_3, \{\})\})\})$$

Note that what has happened here is essentially that we, by a sharing constraint, have *added a component* (`NotDangle`) to the structure $(m_1, \{\})$.

Notice the differece between the two examples. In the latter it was possible to add a component to a structure, while in the first it was impossible. We therefore refer to structures like `Empty` as *rigid* and to structures like `Empty'` occuring inside signatures as *flexible*. Note that the distinction is conceptually very natural; the only reason we have to make use of the concepts rigid and flexible is that structure declarations and structure specifications in the static semantics of SML both elaborate to structures.

3.2 Cover

In HOF it is possible to specify functors inside signatures. Therefore one can specify sharing in a functor signature not only with rigid structures as in Standard ML, but also with flexible structures. In this section we discuss the implications of this.

Consider the following signature expression which is taken from [16].

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

We certainly want that the argument and result signatures of the functor `F` elaborate to principal signatures. Now imagine that we elaborate the signature expression from left to right and use the existing definition of principal signature in Standard ML [12, Section 5.13]. If we assume that the names of the structures for `X` and `X'` are not chosen to be equal when elaborating the specifications of `X` and `X'`, then the basis B will satisfy

$$B(\mathtt{X}) = (m_1, \{\}) \quad B(\mathtt{X}') = (m_2, \{\mathtt{D} \mapsto (m_3, \{\})\})$$

Then the signature for the argument signature expression of `F` is not covered by the basis B (as the structure for `X` has no `D` component) and hence the above signature expression would be rejected. On the other hand, if we had chosen the names in the basis differently, such that B satisfied

$$B(\mathtt{X}) = (m_1, \{\}) \quad B(\mathtt{X}') = (m_1, \{\mathtt{D} \mapsto (m_3, \{\})\})$$

then the signature for the argument signature expression would be covered by the basis and the above signature expression would elaborate. So if we use the definition of cover and principality from Standard ML, a signature expression will (sometimes) only elaborate if we identify names without having met a sharing specification explicitly requiring equality of the names. This would imply loss of principal signatures (just imagine we insert a structure `X'''` in the beginnning of the signature expression above — then we could choose to identify the name of the `X` with either the name of `X'` or the name of `X'''` and the choice would be arbitrary).

Let us continue with the assumption that the names of `X` and `X'` have not been identified in B. Intuitively, the argument signature expression *should* elaborate, namely to

$$(N)S = (\{m_4\})(m_4, \{\texttt{X''} \mapsto (m_1, \{\texttt{D} \mapsto (m_6, \{\})\})\})$$

because `X''` via `X` must share with `X'` which also has a `D` component. This way it is possible to achieve consistency when we meet the last sharing specification and unify the names m_2 and m_1 (and the names m_3 and m_6). Note that if we quantified the name m_6 in the signature, we would have to map the bound name m_6 to the free name m_3 to achieve consistency when meeting the last sharing specification, but bound names must be mapped to bound names — otherwise the fundamental Realisation Theorem, Commentary Section 10.3, does not hold and then one loses principality.

So we would like to be able to express that $(N)S$ is principal, but as we have seen above $(N)S$ is only principal relatively to the choice of the name of the dangling specification inside S (above we chose the name m_6 but we could as well have chosen another name for the `D` component). Moreover, we should record in some way that a component is added to the structure $(m_1, \{\})$ in order to make it possible to achieve consistency when we during elaboration meet a sharing specification. Both these desires can be fulfilled by recording during elaboration the maximal view of any structure that partake in elaboration in a so-called *assembly*. Every time a component is added to a flexible structure it is recorded in the assembly, which therefore will always be the maximal view of any structure.

In the discussion above we have only considered specification of structures and functors in order to keep things as simple as possible As it is also possible to specify sharing between types, the maximal view of any type structure should also be recorded in the assembly as can be seen from the following example.

```
sig
```

```
    type t
    functor F(X:
      sig
        datatype t' = C of int
        sharing type t = t'
      end): sig end
    datatype t'' = C of int | D of int
    sharing type t = t''
  end
```

which should not elaborate (as the type structures for `t'` and `t''` are not consistent). By recording the maximal view of the type structure for `t` we can easily detect the consistency violation when we meet the last sharing specification.

So the assembly should be the maximal view of any structure or type structure that partake in the elaboration of a signature expression. Therefore it is natural to require in the inference rules that the assembly covers the basis. Thus we should be able to distinguish between rigid and flexible structures in an assembly, as it should only be possible to add components to flexible structures (for maximal compatibility with the Definition.)

That requirement, however, implies that our semantics will not be completely compatible with the SML semantics as the following example shows (the first signature in the example is taken from [15]).

```
structure Empty = struct end;
signature SIG =
  sig
    local
      structure NonEmpty :
        sig
          structure Dangle : sig end
        end
      sharing Empty = NonEmpty
    in
    end
  end;
```

```
datatype t = C of int;
signature SIG' =
  sig
    local
      datatype t' = C of int | D of int
      sharing type t = t'
    in
    end
  end;
```

The signature expressions for `SIG` and `SIG'` elaborate in SML (because of the use of `local`), but not in our semantics. The author agrees with Tofte [15] that examples like this are pathological and have no practical significance and therefore accept that our extension in this respect in not a completely conservative extension of SML.

3.3 Principal Signatures

Computation of a principal signature for a signature expression *sigexp* proceeds (as usual in polymorphic type disciplines) by computing the structure S for *sigexp*, and then quantifying a certain subset N of the names free in the structure. How can we determine the subset of names to quantify? Let A be the current assembly before we start elaboration of *sigexp*, and let A' be the assembly after the elaboration of *sigexp*. When we elaborate *sigexp*, componenents can be added to some of the structures and type structures in A. Moreover, A' can contain structures and type structures which are not related at all to the structures and type structures in A. We learned from the earlier section that a name in S which is either free in A or is one of the names we have chosen for the components added to structures or type structures in A should not be quantified. All other names free in S should be quantified. Let N' be the set of names which should not be quantified. As A' is an extension of A the set N' can be obtained as the names of all those structures and type structures occuring in A' which either occurs in A or is a component added to a structure in A. Later we will define an operation Below which given A' and the set of free names in A can compute the structures and type structures occuring in A' which either occurs in A or is a component added to a structure in A. Given this operation Below we can let $A_0 = \mathrm{Below}(A', \mathrm{names}(A))$ and then compute the principal signature as $\mathrm{Clos}_{\mathrm{names}\,A_0} S = (\mathrm{names}\,S \setminus \mathrm{names}\,A_0)S$.

Now consider the following signature expression.

```
sig
  structure S0 : sig end
  functor F1(X:
    sig
      structure S1 : sig end
      structure S2 :
        sig
          functor F2(Y:
            sig
              structure S3 : sig end
              sharing S3 = S1
            end): sig end
        end
      sharing S2 = S0
      structure S1 : sig end
    end): sig end
end
```

Let $sigexp_1$ be the argument signature expression of `F1`. Let us elaborate the above signature expression from left to right in an empty basis and empty assembly. We bind `S0` to the structure $(m_0, \{\})$ and proceed with elaborating $sigexp_1$. Note that the assembly A now contains the structure $(m_0, \{\})$ (and nothing more). We bind `S1` to the structure $(m_x, \{\})$ and `F2` to the functor signature $\Phi = (\{m_2\})((m_2, \{\mathtt{S3} \mapsto (m_x, \{\})\}), \Sigma)$ where $\Sigma = (\{m_{22}\}, (m_{22}, \{\}))$. Because of the sharing specification between `S2` and `S0` we bind `S2` to the structure $(m_0, \{\mathtt{F2} \mapsto \Phi\})$. Then we (re)bind `S1` to the structure $(m_1, \{\})$. Now we have the following structure for $sigexp_1$

$$S = (m_{11}, \{\mathtt{S1} \mapsto (m_1, \{\}), \mathtt{S2} \mapsto (m_0, \{\mathtt{F2} \mapsto \Phi\})\})$$

and we have to determine which names to quantify (as it is required that the *sigexp* elaborates to a principal signature as it is the argument of a functor.) The assembly, call it A', is now equal to A extended with S. Using the operation Below and the closure rule from above, the names in Φ will not be quantified — we will only quantify the names $N = \{m_1, m_{11}\}$.

However, the signature $(N)S$ is not principal as it should be. The reason is, that the name m_x has not been quantified and we are free to choose any name for m_x (As usual in polymorphic type disciplines, if one wants a principality theorem then it should always be the case that, for every name, either the name is quantified or there is only one possible choice for the name.) It is easy to see that the problem is the repeated specification of the structure `S1`. However, the problem does not simply disappear by prohibiting repeated specifations as an example showing the same kind of defect as the above can be constructed using `local` specifications.

We choose to solve the problem by allowing quantification of the name m_x. This of course requires that we change the definition of Below such that Below ignores added functor components. But notice that the quantification of m_x above leads to an ill-formed signature (as the bound name m_x occurs below a free name m_0.) Well, we simply choose to change the definition of well-formedness such that well-formedness of a structure, signature or functor signature only requires that no bound name occurs below a free name, except if it is a name in a functor signature. (See the definition of well-formedness in Section 4.3.) It seems perhaps rather drastic to change the definition of well-formedness, but notice that it has no implications for the first-order part of HOF, so it is still compatible with Standard ML. Technically, the problem manifests itself in the proof of Lemma 6.2.8 in Chapter 6; without the change in the definition of well-formedness, the lemma does not hold.

This solution was also chosen in [16] but in [15] another solution was chosen. The solution was not to quantify m_x but instead to make sure that there will be only one choice of name for m_x. Formally, this was achieved by a strict form of consistency requiring that if two structures share and they both have a functor component F, then the functor signatures for F must be exactly the same in the two structures (this change was also sufficient to prove the above mentioned Lemma 6.2.8). Imagine that `F1` from the above example is applied to a structure `X'`. Then one would create a view of `X'` according to the signature `X` by the usual rules of signature matching. The view would share with `X'`, so by strict consistency the functor components in `X'` and the view should be exactly the same, i.e. there would only be one acceptable choice for the name m_x, namely the name of the `S3` component in the actual structure argument `X'`. This solution is bad because it implies that one cannot even apply the functor `F1` to structures `X'` and `X''` which only differ in the choice of name for the `S3` component (assuming, of course, that they both have a `S3` component). So if one declares structures by

```
structure X'  = strexp
structure X'' = strexp
```

in a functor body of a functor `F` whose argument signature specifies a functor `F'` then one sometimes will only be able to apply the specified functor `F'` on one of the structures inside the body of `F` because structure expressions are generative in SML. This is of course unfortunate, and is avoided in our solution described above.

3.4 Equality Principality

In SML principal signatures must be (c.f. rule 65 in [12]) type explicit [12, Section 5.8] and equality principal [12, Section 5.13].

One of the most fundamental theorems used in proving the existence of principal signatures is the Realisation Theorem, Commentary Section 10.3, which states that elaboration of signature expressions is closed under realisation. We would like to be able to prove a theorem like the Realisation Theorem (note that it is not obvious that one can prove such a theorem for HOF signature expressions as we can have principal signatures inside signatures in HOF) as the author cannot see how to prove principality without such a theorem.[1] However, the Realisation Theorem does not hold for equality-principal signatures (note that the Realisation Theorem in the Commentary is not concerned with signature expression which are elaborated using rule 65).

Let us explain why equality-principality is not preserved under realisation. Assume the following signature expression is required to elaborate to an equality-principal signature.

```
sig
  type t0
  functor F(X: sig
                 datatype t1 = C of int -> int
                 sharing type t1 = t0
               end): sig end
  eqtype t2
  sharing type t0 = t2
```

[1] In fact, I have never seen a proof of principality in polymorphic type disciplines where a theorem like the Realisation Theorem was not used.

```
end
```

Let us elaborate it from left to right. `t0` is elaborated to the type structure $(t', \{\})$, `X` is bound to the signature $\{m_0, t'\}(m_0, \{\texttt{t1} \mapsto (t', \{\texttt{C} \mapsto (\texttt{int} \rightarrow \texttt{int}) \rightarrow t'\})\})$ where t' does not admit equality. Then `t2` is elaborated to the type structure $(t'', \{\})$ where t'' admits equality. We then consider the sharing equation and apply a realisation φ for which $\varphi t' = t''$ on the elaboration tree. The signature for `X` then becomes $\{m_0, t''\}(m_0, \{\texttt{t1} \mapsto (t'', \{\texttt{C} \mapsto (\texttt{int} \rightarrow \texttt{int}) \rightarrow t''\})\})$ but this signature does not respect equality and hence is not equality principal. There is also a problem the other way round: assume the signature expression

```
sig
  type t0
  functor F(X: sig
                 datatype t1 = C of t0
               end): sig end
  eqtype t2
  sharing type t0 = t2
end
```

must elaborate to an equality principal signature. We elaborate `t0` to the type structure $(t', \{\})$ where t' does not admit equality. Then `X` is elaborate to the signature $\{m_0, t''\}(m_0, \{\texttt{t1} \mapsto (t'', \{\texttt{C} \mapsto t' \rightarrow t''\})\})$ where t'' does not admit equality (as t' does not admit equality). `t2` is elaborated to $(t''', \{\})$ where t''' admits equality. Due to the sharing specification we apply a realisation φ for which $\varphi t' = t'''$ and get the following signature for `X`: $\{m_0, t''\}(m_0, \{\texttt{t1} \mapsto (t'', \{\texttt{C} \mapsto t''' \rightarrow t''\})\})$ and now t'' should admit equality in order for this signature to be equality principal but t'' does not admit equality.

The problem has to do with maximization of equality in a "flexible context", that is in a context where we do not have sufficient information concerning the equality of type names. We choose to solve the problem by deferring maximization of equality and check for equality principality. Instead of maximizing when we form a principal signature we maximize equality when we are to bind a signature identifier to a signature (at the syntactic level of signature bindings). The implementation of maximization of equality is described in Section 8.13.

3.5 Type Explication

Type-explication is used to ban certain signatures. The idea is that for type-explicit signatures, there is at most one realisation φ such that a structure matches a signature via φ, see Theorem 7.1 in the Commentary. We are not exactly concerned with signature matching here so we choose, for simplicity, to check for type explication when a signature identifier is bound to a signature instead of when a signature is formed. As discussed in the Commentary, Page 113, it is not clear whether or not type explication should be part of admissibility, and in fact we do impose a kind of type explication on assemblies.

Chapter 4

Static Semantics for HOF

In this chapter we present the static semantics of HOF based on the disussion in the preceding chapter. The form of the presentation follows the Definition closely.

4.1 Semantic Objects

The semantic objects[1] for HOF static semantics are almost as for Standard ML, see Figures 9–11 in the Definition. The definition of Env is changed, since environments now also must have a functor environment component because functors can occur inside structures. Moreover, we define a new semantic object, Asmb, since we will manipulate assemblies directly in the inference rules. The changed and new semantic objects are shown in Figure 4.1. An assembly can be thought of as a list of structures and type

$$
\begin{array}{rcl}
E \text{ or } (F, SE, TE, VE, EE) & \in & \mathrm{Env} = \mathrm{FunEnv} \times \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv} \times \mathrm{ExConEnv} \\
A & \in & \mathrm{Asmb} = \mathrm{EmptyAsmb} \cup \mathrm{Str} \times \mathrm{Asmb} \cup \mathrm{TyStr} \times \mathrm{Asmb} \\
 & & \mathrm{EmptyAsmb} = \{\epsilon\}
\end{array}
$$

Figure 4.1: Further Compound Semantic Objects for HOF

structures. The definition of Asmb is like the definition of an assembly in [16]. However, as we consider the full language, our assembly is not only a list of structures but also includes type structures, c.f. Chapter 3.

[1]By semantic objects we simply mean mathematically defined objects as in the Definition. Notice that other authors have different opinions on what semantic objects are, see for example [7].

Besides the operations on semantics objects defined in [12, Section 5.1] we define a function "skel" (for skeleton) on structures, environments, structure environments, functor environments, type environments, variable environments, exception environments, and constructor environments recursively as follows:

$$
\begin{array}{lcl}
\mathrm{skel}(m, E) & = & (m, \mathrm{skel}(E)) \\
\mathrm{skel}(F, SE, TE, VE, EE) & = & (\mathrm{skel}(F), \mathrm{skel}(SE), \mathrm{skel}(TE), \{\}, \{\}) \\
\mathrm{skel}(\{strid_1 \mapsto S_1, \ldots, strid_k \mapsto S_k\}) & = & \{strid_1 \mapsto \mathrm{skel}(S_1), \ldots, strid_k \mapsto \mathrm{skel}(S_k)\} \\
\mathrm{skel}(\{funid_1 \mapsto \Phi_1, \ldots, funid_k \mapsto \Phi_k\}) & = & \{funid_1 \mapsto \Phi_0, \ldots, funid_k \mapsto \Phi_0\} \\
\mathrm{skel}(\{tycon_1 \mapsto (\theta_1, CE_1), \ldots, tycon_k \mapsto (\theta_k, CE_k)\}) & = & \\
\quad \{tycon_1 \mapsto (\theta_1, \mathrm{skel}(CE_1)), \ldots, tycon_k \mapsto (\theta_k, \mathrm{skel}(CE_k))\} & & \\
\mathrm{skel}(\{con_1 \mapsto \sigma_1, \ldots, con_k \mapsto \sigma_k\}) & = & \{con_1 \mapsto \sigma_0, \ldots, con_k \mapsto \sigma_0\}
\end{array}
$$

where $\Phi_0 = (\{\mathtt{m1}\})((\mathtt{m1}, \{\}), (\{\mathtt{m2}\})(\mathtt{m2}, \{\}))$ (an arbitrary functor signature with no free names) and σ_0 is an arbitrary closed (with no free names) type scheme. This operation, skel, will be used to exempt functor signatures from semantic objects as needed in the definition of well-formedness and of Below, c.f. Chapter 3. Moreover, it is used to exempt type schemes in constructor environments as we are only interested in the domains of constructor environments (to check consistency of type structures, c.f. the definition of consistency and cover below). The notion of type explication to be introduced later means that we can exempt variable and exception environments.

A structure (m, E) *occurs free* in a semantic object A if (m, E) occurs in A and m is free. A type structure (θ, CE) occurs free in a semantic object A if (θ, CE) occurs in A and every type name in θ is free.[2]

4.2 Consistency

We use exactly the same definition of constistency as in [12].

4.3 Well-formedness

We keep the definition of type structure well-formedness [12, Section 4.9], but, as explained in the preceding chapter, change the definition of well-formedness for the semantic objects of the HOF static semantics as follows.

[2]This definition is almost the same as the definition in the Commentary, Appendix A, Page 117 — the only change is that we do not require that A is admissible.

Definition 4.3.1 (Well-formedness) *A signature* $(N)S$ *is* well-formed *if* S *is well-formed,* $N \subseteq \mathrm{names}\, S$*, and also, whenever* (m, E) *occurs free in* S *and* $m \notin N$*, then* $N \cap \mathrm{names}(\mathrm{skel}\, E) = \emptyset$*. A functor signature* $(N)(S, (N')S')$ *is* well-formed *if* $(N)S$ *and* $(N')S'$ *are well-formed, and also, whenever* (m', E') *occurs free in* S' *and* $m' \notin N \cap N'$*, then* $(N \cup N') \cap \mathrm{names}(\mathrm{skel}\, E') = \emptyset$*. A semantic object* A *is* well-formed *if every type structure, signature and functor signature occuring in* A *is well-formed.*

Note that the skel function, besides exempting functor components as discussed in Chapter 3, also exempts variable and exception environments. This means that this definition of well-formedness is less restrictive than the definition of well-formedness in the Definition [12, Section 5.3]: a bound type name can occur below a free name if it occurs in the variable or exception environment. But type explication (see the definition below) ensures that for any bound type name t there exists a type structure (t, CE) for some CE in the range of a type environment and this type name is not exempted and hence cannot occur below a free name due to well-formedness (contradiction). So type explication ensures that we get the same properties as in SML in the end.

4.4 Cycle-freedom

Cycle-freedom of a semantic object is defined as in the Definition.

4.5 Cover

Definition 4.5.1 (Cover) *Let* A_1 *and* A_2 *be semantic objects, and let* N *be a name set. We say that* A_2 covers A_1 on N *if* $N \cap \mathrm{names}(A_1) \subseteq \mathrm{names}(A_2)$ *and also, for all* (m, E_1)*, if* (m, E_1) *occurs free in* A_1 *and* $m \in N$ *then*

1. *for all* $strid \in \mathrm{Dom}\, E_1$ *there exists an* E_2 *such that* (m, E_2) *occurs free in* A_2 *and* $strid \in \mathrm{Dom}\, E_2$
2. *for all* $funid \in \mathrm{Dom}\, E_1$ *there exists an* E_2 *such that* (m, E_2) *occurs free in* A_2 *and* $funid \in \mathrm{Dom}\, E_2$
3. *for all* $tycon \in \mathrm{Dom}\, E_1$ *there exists an* E_2 *such that* (m, E_2) *occurs free in* A_2 *and* $tycon \in \mathrm{Dom}\, E_2$

and also, for all (θ, CE_1)*, if* (θ, CE_1) *occurs free in* A_1 *and* $\theta \in N$ *then there exists a* CE_2 *such that* (θ, CE_2) *occurs free in* A_2 *and* $\mathrm{Dom}\, CE_1 = \mathrm{Dom}\, CE_2$ *or* $\mathrm{Dom}\, CE_1 = \emptyset$*. We say that* A_2 covers A_1 *if* A_2 *covers* A_1 *on* names(A_1)*. We say* A_2 *is a* conservative cover *of* A_1 *if* A_2 *covers* A_1 *and* A_1 *covers* A_2 *on* names(A_1)*.*

This definition is used to ensure that the maximal view of any structure or type-structure will be recorded in the assembly during elaboration — if A_2 covers A_1 then for all structures and type-structures in A_1, there is a larger (with more components) in A_2. Conservative cover is used to express that it should be impossible to add components to rigid structures, c.f. Section 3.2.

4.6 Admissibility

Definition 4.6.1 (Admissibility) *A semantic object or a collection of semantic objects* A *is admissible if it is cycle-free, consistent and well-formed. We say that* A_2 *is* an admissible cover *of* A_1*, written* $A_1 \sqsubseteq A_2$*, if* (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* (A_1, A_2) *is admissible and* A_2 *is a conservative cover of* A_1*. Both* $\sqsubseteq$ *and* $\trianglelefteq$ *are preorders. Note that* $A_1 \trianglelefteq A_2$ *implies* $A_1 \sqsubseteq A_2$*. We say that* A_1 *and* A_2 *are* equivalent*, written* $A_1 \sim A_2$*, if* $A_1 \sqsubseteq A_2$ *and* $A_2 \sqsubseteq A_1$*.*

4.7 Type Realisation

The definitions of type realisation and the support of a type realisation are as in the Definition.

4.8 Realisation

The definitions of realisation, support and yield of a realisation are as in the Definition. If N is a name set, then $\varphi(N)$ means $\{\, \varphi(n) \mid n \in N \,\}$.

4.9 Type Explication

Definition 4.9.1 *A signature $(N)S$ is* type-explicit *if every functor signature occurring in S is type-explicit and if, whenever $t \in N$ and occurs free in S, then some substructure of S contains a type environment TE such that $TE(tycon) = (t, CE)$ for some $tycon$ and some CE. A functor signature $(N)(S, (N')S')$ is $(N)S$ is type-explicit. An assembly A is type-explicit if, for all $t \in \mathrm{tynames}\, A$ there exists a CE such that (t, CE) occurs free in A.*

Type explication of a functor signature is defined to be compatible with the Definition, c.f. the discussion in the Commentary [11, Page 113]. Type explication of an assembly simply expresses that type names in an assembly do not "come out of the blue". If a variable, for instance, is specified to have a certain type name, then that type name has been specified or declared earlier on and as an assembly should contain the maximal view of any structure and type structure partaking in the elaboration there must be some type structure corresponding to the type name in the assembly.

4.10 Signature Instantiation

The definition of signature instantiation is as in the Definition.

4.11 The Category K

Recall that we will manipulate assemblies in the inference rules, and that the assembly invariantly is going to record the maximal view of any structure or type-structure. To get a theorem like the Realisation Theorem of the Commentary [11, Page 99], it must be possible to express that components can be added to flexible structures when two structures or type-structures not sharing "starts to share". It is not easy to model adding components by substitution as known from work on polymorphic record typing [14], but it is in fact possible to express the widening of structures using a record type discipline [2]. Here, however, we follow Tofte [16] and consider φ to be a relation between semantic objects, expressed via elementary category theory.

Definition 4.11.1 (Category K) *K is the category defined as follows. An object O of K is a pair (A, B), where A is a type-explicit assembly and $B = (N, F, G, E)$ is a basis satisfying that $A \sqsupseteq B$ (by which we mean $A \sqsupseteq (F, G, E)$ and $\mathrm{names}(A) \supseteq N$). The set of objects of K is denoted* Obj. *For all objects $O_1 = (A_1, B_1) = (A_1, (N_1, F_1, G_1, E_1))$ and $O_2 = (A_2, B_2)$, and for every 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. 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 $n \in \mathrm{names}\, O_1$. Composition in K is the natural extension of composition of realisations.

The condition "A_1 covers A_2 on N_1" prevents realisation from adding components to any structure or type-structure, whose name is in N_1, i.e. the rigid structures. Notice that $O_1 \xrightarrow{\varphi} O_2$ implies $\varphi(B_1) = \varphi(N_1, F_1, G_1, E_1) = (N_1, \varphi F_1, \varphi G_1, \varphi E_1) = B_2$, i.e. components are not added to structures recorded in the basis.

Note that, by the definition of composition of morphisms in K, commutativity of a diagram of the form

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

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

4.12 Principal Signatures

Let B be a basis, A an assembly and let *sigexp* be a signature expression. In Section 4.13 we shall define what it is for *sigexp* to elaborate to a structure S in (A, B), written $A, B \vdash sigexp \Rightarrow S$. This is used in the definition of principal signature[3]:

Definition 4.12.1 (Principal Signature) *We say that a signature $\Sigma = (N)S$ is* principal *for sigexp in $O = (A, B)$ if $O \in \mathrm{Obj}$ and, choosing N such that $N \cap \mathrm{names}(A, B) = \emptyset$*

1. *There exists an A' such that $A \trianglelefteq A'$ and $A', B \vdash sigexp \Rightarrow S$*

2. *For all O', φ and S', if $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow S'$ then $\varphi((N)S) \geq S'$*

Conservative cover is needed in item 1 (it is not enough to use $A \sqsubseteq A'$ instead of $A \trianglelefteq A'$) because otherwise A would not contain the maximal view of any structure or type-structure partaking in the elaboration of the signature expression.

Item 2 expresses that the principal signature Σ is general enough that no matter how the flexible structures in A are widened, all possible results of elaborating the *sigexp* can be obtained by instantiation of Σ.

[3]Recall that $\Sigma \geq S$ is used to express that S is an instance of Σ — c.f. Definition, Section 5.9.

The definition of when a semantic object respects equality, and the definition of equality-principal signature is as in the Definition (of course, in terms of the above new definition of principal signature).

4.13 Inference Rules

In this section we present the inference rules for signature bindings and signature expressions. We have not included the inference rules for type expressions and type-expression rows as these are exactly as in the Definition [12, Rules 47–52].

All the conclusions of the rules are of the form

$$P \triangleright \mathit{phrase} \Rightarrow Q$$

where *phrase* is one of the phrase-forms defined in the grammar (Section 2.1), and P and Q are semantic objects.

In the premises of the rules, "$A, B \vdash \mathit{phrase} \Rightarrow Q$" is used as a macro meaning "$(A, B) \in \mathrm{Obj}$, $A \sqsupseteq Q$ and $A, B \triangleright \mathit{phrase} \Rightarrow Q$". Since $(A, B) \in \mathrm{Obj}$ simply means that A is type-explicit and $A \sqsupseteq B$ this is equivalent to "$A \sqsupseteq (B, Q)$ and $A, B \triangleright \mathit{phrase} \Rightarrow Q$ and A type-explicit".

We say that *phrase elaborates to* Q *in* (A, B), written $A, B \vdash \mathit{phrase} \Rightarrow Q$, if there is an inference tree which satisfies all the side-conditions on the rules and proves $A, B \triangleright \mathit{phrase} \Rightarrow Q$ and also $(A, B) \in \mathrm{Obj}$ and $A \sqsupseteq Q$. We say that *phrase elaborates to* Q *in* C, written $C \vdash \mathit{phrase} \Rightarrow Q$, if there is an inference tree which satisfies all the side-conditions on the rules and proves $C \triangleright \mathit{phrase} \Rightarrow Q$ and also (C, P) is admissible.

Signature Expressions $\boxed{A, B \triangleright \mathit{sigexp} \Rightarrow S}$

$$\frac{A, B \vdash \mathit{spec} \Rightarrow E}{A, B \triangleright \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)} \tag{4.1}$$

$$\frac{B(\mathit{sigid}) \geq S}{A, B \triangleright \mathit{sigid} \Rightarrow S} \tag{4.2}$$

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

$$\frac{\begin{array}{l} A \trianglelefteq A' \qquad A', B \vdash sigexp \Rightarrow S \\ (N)S \text{ principal for } sigexp \text{ in } A, B \\ N \cap \text{names}\, A = \emptyset \end{array}}{A, B \rhd sigexp \Rightarrow (N)S} \tag{4.3}$$

Specifications

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

$$\frac{C \text{ of } B \vdash valdesc \Rightarrow VE}{A, B \rhd \texttt{val}\ valdesc \Rightarrow \text{Clos}VE \text{ in Env}} \tag{4.4}$$

$$\frac{C \text{ of } B \vdash typdesc \Rightarrow TE}{A, B \rhd \texttt{type}\ typdesc \Rightarrow TE \text{ in Env}} \tag{4.5}$$

$$\frac{C \text{ of } B \vdash typdesc \Rightarrow TE \qquad \forall(\theta, CE) \in \text{Ran}\, TE,\ \theta \text{ admits equality}}{A, B \rhd \texttt{eqtype}\ typdesc \Rightarrow TE \text{ in Env}} \tag{4.6}$$

$$\frac{C \text{ of } B + TE \vdash datdesc \Rightarrow VE, TE}{A, B \rhd \texttt{datatype}\ datdesc \Rightarrow (VE, TE) \text{ in Env}} \tag{4.7}$$

$$\frac{C \text{ of } B \vdash exdesc \Rightarrow EE \quad VE = EE}{A, B \rhd \texttt{exception}\ exdesc \Rightarrow (VE, EE) \text{ in Env}} \tag{4.8}$$

$$\frac{A, B \vdash strdesc \Rightarrow SE}{A, B \rhd \texttt{structure}\ strdesc \Rightarrow SE \text{ in Env}} \tag{4.9}$$

$$\frac{A, B \vdash fundesc \Rightarrow F}{A, B \rhd \texttt{functor}\ fundesc \Rightarrow F \text{ in Env}} \tag{4.10}$$

$$\frac{A, B \vdash shareq \Rightarrow \{\}}{A, B \rhd \texttt{sharing}\ shareq \Rightarrow \{\} \text{ in Env}} \tag{4.11}$$

$$\frac{A, B \vdash spec_1 \Rightarrow E_1 \qquad A, B + E_1 \vdash spec_2 \Rightarrow E_2}{A, B \rhd \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow E_2} \tag{4.12}$$

$$\frac{B(longstrid_1) = (m_1, E_1) \quad \cdots \quad B(longstrid_n) = (m_n, E_n)}{A, B \rhd \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n} \tag{4.13}$$

$$\frac{B(sigid_1) \geq (m_1, E_1) \quad \cdots \quad B(longstrid_n) \geq (m_n, E_n)}{A, B \triangleright \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow E_1 + \cdots + E_n} \tag{4.14}$$

$$\frac{}{A, B \triangleright \qquad \Rightarrow \{\}} \tag{4.15}$$

$$\frac{A, B \vdash spec_1 \Rightarrow E_1 \qquad A, B + E_1 \vdash spec_2 \Rightarrow E_2}{A, B \triangleright spec_1\ \langle ; \rangle\ spec_2 \Rightarrow E_1 + E_2} \tag{4.16}$$

Value Descriptions

$\boxed{C \triangleright valdesc \Rightarrow VE}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE \rangle}{C \triangleright var\ :\ ty\ \langle \texttt{and}\ valdesc \rangle \Rightarrow\ \{var \mapsto \tau\}\ \langle +VE \rangle} \tag{4.17}$$

Type Descriptions

$\boxed{C \triangleright typdesc \Rightarrow TE}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad \langle C \vdash typdesc \Rightarrow TE \rangle \qquad \text{arity } \theta = k}{C \triangleright tyvarseq\ tycon\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \{tycon \mapsto\ (\theta, \{\})\}\ \langle +VE \rangle} \tag{4.18}$$

Datatype Descriptions

$\boxed{C \triangleright datdesc \Rightarrow (VE, TE)}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)}t \vdash condesc \Rightarrow CE \qquad \langle C \vdash datdesc \Rightarrow VE, TE \rangle}{\begin{array}{c} C \triangleright tyvarseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \text{Clos}CE \langle +VE \rangle,\ \{tycon \mapsto (t, \text{Clos}CE)\}\ \langle +TE \rangle \end{array}} \tag{4.19}$$

Constructor Descriptions

$\boxed{C, \tau \triangleright condesc \Rightarrow CE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash condesc \Rightarrow CE \rangle\rangle}{\begin{array}{l} C, \tau \triangleright con\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{|}\ condesc \rangle\rangle \Rightarrow \\ \qquad \{con \mapsto \tau\}\ \langle +\{con \mapsto \tau' \to \tau\} \rangle\ \langle\langle +\ CE \rangle\rangle \end{array}} \tag{4.20}$$

Exception Descriptions

$\boxed{C \triangleright exdesc \Rightarrow EE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau \qquad \text{tyvars}(ty) = \emptyset \rangle \qquad \langle\langle C \vdash exdesc \Rightarrow EE \rangle\rangle}{\begin{array}{l} C \triangleright excon\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exdesc \rangle\rangle \Rightarrow \\ \qquad \{excon \mapsto \texttt{exn}\}\ \langle +\ \{excon \mapsto \tau \to \texttt{exn}\} \rangle\ \langle\langle +\ EE \rangle\rangle \end{array}} \tag{4.21}$$

Structure Descriptions

$$\boxed{A, B \rhd strdesc \Rightarrow SE}$$

$$\frac{A, B \vdash sigexp \Rightarrow S \qquad \langle A, B \vdash strdesc \Rightarrow SE\rangle}{A, B \rhd strid\ \texttt{:}\ sigexp\ \langle \texttt{and}\ strdesc\rangle \Rightarrow \{strid \mapsto S\}\ \langle +\ SE\rangle} \tag{4.22}$$

Functor Descriptions

$$\boxed{A, B \rhd fundesc \Rightarrow F}$$

$$\frac{A, B \vdash funsigexp \Rightarrow \Phi \qquad \langle A, B \vdash fundesc \Rightarrow F\rangle}{A, B \rhd funid\ funsigexp\ \langle \texttt{and}\ fundesc\rangle \Rightarrow \{funid \mapsto \Phi\}\ \langle +\ F\rangle} \tag{4.23}$$

Sharing Equations

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

$$\frac{m \text{ of } B(longstrid_1) = \cdots = m \text{ of } B(longstrid_n)}{A, B \rhd longstrid_1\ \texttt{=}\ \cdots\ \texttt{=}\ longstrid_n \Rightarrow \{\}} \tag{4.24}$$

$$\frac{\theta \text{ of } B(longtycon_1) = \cdots = \theta \text{ of } B(longtycon_n)}{A, B \rhd \texttt{type}\ longtycon_1\ \texttt{=}\ \cdots\ \texttt{=}\ longtycon_n \Rightarrow \{\}} \tag{4.25}$$

$$\frac{A, B \rhd shareq_1 \Rightarrow \{\} \qquad A, B \rhd shareq_2 \Rightarrow \{\}}{A, B \rhd shareq_1\ \texttt{and}\ shareq_2 \Rightarrow \{\}} \tag{4.26}$$

Functor Signature Expressions

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

$$\frac{\begin{array}{c} A, B \vdash sigexp_1 \Rightarrow\ (N)S \qquad N \cap \text{names}\, A = \emptyset \\ (S, A), B + N + \{strid \mapsto S\} \vdash sigexp_2 \Rightarrow \Sigma \end{array}}{A, B \rhd \texttt{(}\ strid\ \texttt{:}\ sigexp_1\ \texttt{)}\ \texttt{:}\ sigexp_2 \Rightarrow (N)(S, \Sigma)} \tag{4.27}$$

Signature Bindings

$$\boxed{A, B \rhd sigbind \Rightarrow G}$$

$$\frac{\begin{array}{ll} A, B \vdash sigexp \Rightarrow \Sigma & \Sigma \text{ is type-explicit} \\ \langle A, B \vdash sigbind \Rightarrow G\rangle & \Sigma' \text{ equality-principal}(\Sigma) \end{array}}{A, B \rhd sigid\ \texttt{=}\ sigexp\ \langle \texttt{and}\ sigbind\rangle \Rightarrow \{sigid \mapsto \Sigma'\}\ \langle +G\rangle} \tag{4.28}$$

We now comment on those rules which diverge most from the rules in the Definition.

Consider rule 4.3, and recall that the assembly is supposed to hold the maximal view of any structure that partake in elaboration. Then it is natural that when we bind structures that occur in S by (N) then we simultaneously discharge the corresponding structures from the assembly; this is expressed by conservative cover, $A \trianglelefteq A'$. We use *conservative* cover, instead of simply cover, as it should only be those structures that are used locally in the elaboration of *sigexp* that are discharged.

In rule 4.27 the assembly is extended by S during elaboration of $sigexp_2$ as the structure S here "enters" the elaboration — this is just the opposite to the discharging explained above. The side-condition $N \cap \text{names}\, A = \emptyset$ is simply used to avoid accidental sharing. The name set N is added to the set of rigid names in order to ensure that the elaboration of $sigexp_2$ does not add components to a structure whose name is in N (this is to ensure that the *funid* can be applied to any structure that just matches $sigexp_1$, c.f. Example 8.1 in the Commentary). But notice that we use "$+N + \{strid \mapsto S\}$" instead of "$\oplus\ \{strid \mapsto S\}$" as in rule 94 in the Definition. This is crucial; it makes it possible to add components to flexible structures outside the functor by sharing specifications in the body of the functor. For example, the following signature expression would not elaborate if we had used $\oplus$.

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

In rule 4.28 we see the side-conditions about type explication and equality-principality, as discussed in Sections 3.4 and 3.5.

Chapter 5

Elaboration of Signature Expressions

In this chapter we state and prove some propositions concerning the elaboration of signature expressions, most notably the Admissification Theorem and the Realisation Theorem.

5.1 Admissification

Our definition of admissifier is changed compared to the definition in the Commentary. Our definition is taken from [16]. Before presenting the definition of admissifier, we repeat a couple of fundamental definitions from the Commentary in order to ease reading.

For every basis B and name n, n is *rigid with respect to* B, if $n \in N$ of B, and *flexible with respect to* B otherwise. A basis B is *rigid* if names $B \subseteq N$ of B.

Let A be an assembly (not necessarily admissible), let φ be a realisation and N a name set. Then φ is said to be *fixed on* N if $\varphi(n) = n$, for all $n \in N$. Moreover, φ is said to be an *admissifier for* A *under* N if φ is fixed on N, A covers φA on N and φA is admissible. Note that, compared to the definition of admissifier in the Commentary, the only change is that we have added "A covers φA on N" — this is to ensure that admissifiers do not add components to rigid structures, c.f. Section 4.11. An admissifier φ^* is said to be *most general* (or *principal*) if, for every admissifier φ for A under N, there exists a realisation φ' such that φ' is fixed on N and $\varphi'(\varphi^* A) = \varphi A$.

A semantic object A is *grounded in* N if (after disjoining the bound names in A from N) for every occurrence of a type-structure (θ, CE) in A not within a functor signature, either θ is simply a type name, or $\mathrm{tynames}(\theta) \subseteq N$. Further we say a basis B is *grounded* if it is grounded in N of B, and we say that an object $O = (A, B)$ of K is grounded if B is *grounded* and A is grounded in N of B.

The following lemma is taken from the Commentary. The proof is not shown in the Commentary, however, so we provide the proof here.

Lemma 5.1.1 (Inverse Realisation) *Let φ be a realisation, and A an assembly. Then if φA is well-formed, so is A; and if φA is cycle-free, so is A.*

Proof Well-formedness of A: Assume A is not well-formed. Then either (i) there is an ill-formed type-structure (θ, CE) in A, or (ii) there is an ill-formed signature $(N)S$ in A or (iii) there is an ill-formed functor signature Φ in A. Assume (i). Then θ is not a type name and $CE \neq \{\}$, but then θ of $\varphi(\theta, CE)$ is not a type name and CE of $\varphi(\theta, CE) \neq \{\}$ contradicting the well-formedness of φA. Now assume (ii). Then either $N \not\subseteq \mathrm{names}\, S$ or there is a free occurence of a structure (m, E) in S, with $m \notin N$, such that $N \cap \mathrm{names}(\mathrm{skel}\, E) \neq \emptyset$. Assume $N \not\subseteq \mathrm{names}\, S$, and then let $n \in N \setminus \mathrm{names}\, S$. As bound names are changed to avoid name capturing when applying a realisation, it cannot be the case that, for a $n' \in \mathrm{names}\, S$, $\varphi n' = \varphi n$, hence $\varphi n \notin \mathrm{names}(\varphi S)$, contradicting the well-formedness of φA. Then assume (m, E) occurs free in S, with $m \notin N$ and that $N \cap \mathrm{names}(\mathrm{skel}\, E) \neq \emptyset$. Let $n \in N \cap \mathrm{names}(\mathrm{skel}\, E)$. Then $\varphi n \in \varphi N \cap \mathrm{names}(\mathrm{skel}\, \varphi E)$ but φm is free, as name-capturing is avoided, contradicting the well-formedness of φA. Then assume (iii). By an argumentation analogous to case (ii) we get a contradiction of well-formedness of φA. Hence A is well-formed.

Cycle-freeness of A: Assume A is not cycle-free. Then there is a sequence of structure names

$$m_0, \ldots, m_{k-1}, m_k = m_0 \qquad k > 0$$

such that for each $i (0 \leq i < k)$ some structure with name m_i occuring in A has a proper substructure with name m_{i+1}. But then φA would have a cycle

$$\varphi m_0, \ldots, \varphi m_{k-1}, \varphi m_k = m_0$$

contradicting that φA is cycle-free. □

The following lemma is taken from [16]; the proof is an extended version of the proof in [16] as we have to consider type-structures too.

Lemma 5.1.2 (Inverse Cover) *Let φ be a realisation 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 Since names$(\varphi A_2) \cap N \subseteq A_1$, by definition of cover, and φ is fixed on N we have names$(A_2) \cap N \subseteq$ names(A_1) as required. Let (m, E) occur free in A_2, $m \in N$ and let id be a structure or functor identifier or a type constructor such that $id \in \mathrm{Dom}\, E$. 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 $id \in \mathrm{Dom}\, E_0$, as required. Now let (θ, CE) occur free in A_2 and assume $\theta \in N$. Then since φ is fixed on N, $(\theta, \varphi CE)$ occurs free in φA_2 and $\theta \in N$. Since A_1 covers φA_2 on N there exists a CE_0 such that (θ, CE_0) occurs free in A_1. The domain requirements on (CE_0, CE) are satisfied as $\mathrm{Dom}\, CE = \mathrm{Dom}(\varphi CE)$ and the domain requirements of $(CE_0, \varphi CE)$ are satisfied. □

Theorem 5.1.1 (Admissification) *Let A be an arbitrary assembly grounded in a name set N. If there exists some admissifier φ for A under N then there exists a principal admissifier φ^* for A under N. Moreover, $\varphi^* A$ is grounded in N.*

Proof The proof is almost the same as in the Commentary [11, Page 102][1]. However, due to the new definition of admissifier, we need to add the following when proving admissibility of $\varphi^* A$. "Moreover, since $\varphi(\varphi^* A) = \varphi A$ and A covers φA on N we have that A covers $\varphi^* A$ on N by lemma 5.1.2." □

The following lemma is taken from [16] but the proof is extended compared to the proof in [16].

Lemma 5.1.3 (Cover under Realisation Decomposition) *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 5.1.2.

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 type constructor or 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 $id \in \mathrm{Dom}\, E_1$. 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

[1]There is is a minor omission in the proof in the Commentary; replace "$F(\equiv)$ is the least equivalence $\equiv'$ such that" with "$F(\equiv)$ is the least equivalence $\equiv'$ containing $\equiv$ such that".

free in A_2 and $id \in \text{Dom}\, E_2$, as required. Now let (θ, CE) be a type-structure occuring free in A_3 and asume $\theta \in N$. Then as A_1 covers A_3 on N there exists a CE_1 such that (θ, CE_1) occurs free in A_1 and $\text{Dom}\, CE = \{\}$ or $\text{Dom}\, CE = \text{Dom}\, CE_1$. Then, since φ_1 is fixed on N and $\varphi_1 A_1 \sqsubseteq A_2$, $\varphi(\theta, CE_1) = (\theta, \varphi CE_1)$ is covered by A_2. Thus there exists a CE_2 such that (θ, CE_2) occurs free in A_2 and $\text{Dom}\, CE_2 = \text{Dom}(\varphi CE_1) = \text{Dom}\, CE$ or $(\text{Dom}(\varphi CE) = \{\} \Leftrightarrow \text{Dom}\, CE = \{\})$, as required. Thus A_2 covers A_3 on N. □

5.2 The Realisation Theorem

The following lemma is taken from Page 99 in the Commentary.

Lemma 5.2.1 (Instantiation preserved under realisation) *For any signature Σ, structure S and realisation φ, if $\Sigma \geq S$ then $\varphi\Sigma \geq \varphi S$.*

Proof As in the Commentary, Page 99. □

The following lemma is taken from [16], but the proof is extended as we also consider covering of type-structures.

Lemma 5.2.2 (Cover preserved under realisation) *If A_2 covers A_1 then φA_2 covers φA_1.*

Proof We have to prove φA_2 covers φA_1 on names(φA_1). Since names$(A_1) \subseteq$ names(A_2) we have names$(\varphi A_1) \subseteq$ names(φA_2) as required. Let (m, E_1) be a structure occurring free in φA_1, and let id be a type constructor or a structure or functor identifier such that $id \in \text{Dom}\, E_1$. Then $(m, E_1) = \varphi(m', E_1')$, for some (m', E_1') occurring free in A_1. Since $id \in \text{Dom}\, E_1'$ and A_2 covers A_1, there exists an E_2' such that (m', E_2') occurs free in A_2 and $id \in \text{Dom}\, E_2'$. Thus $\varphi(m', E_2') = (m, \varphi E_2')$ occurs free in φA_2 with $id \in \text{Dom}\, \varphi E_2'$, as required. Let (θ, CE_1) be a type-structure occuring free in φA_1. Then $(\theta, CE_1) = \varphi(\theta', CE_1')$ for some (θ', CE_1') occuring free in A_1 and $\text{Dom}\, CE_1 = \text{Dom}\, CE_1'$. Since A_2 covers A_1 there exists a CE_2' such that (θ', CE_2') occurs free in A_2 with $\text{Dom}\, CE_2' = \text{Dom}\, CE_1'$ or $\text{Dom}\, CE_1' = \{\}$. Hence there exists $\varphi(\theta', CE_2') = (\theta, CE_2)$ in φA_2 with $\text{Dom}\, CE_2' = \text{Dom}\, CE_2$, and $\text{Dom}\, CE_2 = \text{Dom}\, CE$ or $\text{Dom}\, CE = \{\}$, as required, proving that φA_2 covers φA_1. □

The following lemma is taken from [16], but the proof is written out in detail as opposed to the proof in [16].

Lemma 5.2.3 (Admissible Cover preserved under realisation) *If $A_1 \sqsubseteq A_2$ and φA_2 is admissible, then $\varphi A_1 \sqsubseteq \varphi A_2$.*

Proof We have that φA_2 covers φA_1 by Lemma 5.2.2. It remains to prove that $(\varphi A_1, \varphi A_2)$ is admissible.

Well-formedness of type-structures: We have to show that for all (θ, CE) occuring in $(\varphi A_1, \varphi A_2)$, $CE = \{\}$ or $\theta \in$ TyName. It holds if (θ, CE) occurs in φA_2 as φA_2 is admissible; therefore assume (θ, CE) occurs in φA_1. Assume θ is not a type name (if it is, well-formedness holds trivially). Then as φA_2 covers φA_1, there exists a type structure (θ, CE') occuring in φA_2, such that $CE = \{\}$ or $\mathrm{Dom}\, CE = \mathrm{Dom}\, CE'$. If $CE = \{\}$ then well-formedness is satisfied; if $\mathrm{Dom}\, CE = \mathrm{Dom}\, CE'$ then as φA_2 is admissible and θ is not a type name $\mathrm{Dom}\, CE' = \{\}$, hence $\mathrm{Dom}\, CE = \{\}$, proving well-formedness of type-structures in φA_1.

Well-formedness of signatures: Let $(N)S$ be a signature occuring in φA_1 (if none exists, well-formedness is satisfied) and assume S is well-formed. Then there exists $(N')S'$ occuring in A_1 such that $\varphi((N')S') = (N)S$. As $(N')S'$ is well-formed, $N' \subseteq \mathrm{names}\, S'$ and therefore $N \subseteq \mathrm{names}\, S$, as required. Now let (m, E) be a structure occuring free in S (if none exists well-formedness holds) and $m \notin N$. Then there exists (m', E') occuring in A_1 and $m' \notin N'$. Then as A_1 is well-formed $N' \cap \mathrm{names}(\mathrm{skel}\, E') = \emptyset$, hence $N \cap \mathrm{names}(\mathrm{skel}\, E) = \emptyset$, as name capture is avoided when applying a realisation, proving well-formedness of signatures.

Well-formedness of functor signatures: Let $(N)(S, (N')S')$ be a functor signature occuring in A_1 (if none exists, well-formedness is satisfied) and assume $(N)S$ and $(N')S'$ are well-formed. Then there exists $(N_0)(S_0, (N_0')S_0')$ occuring in A_1 such that $\varphi((N_0)(S_0, (N_0')S_0')) = (N)(S, (N')S')$. Now let (m', E') occur free in S' with $m' \notin N \cup N'$ (if none exists, well-formedness is satisfied). Then there exists (m_0', E_0') occuring free in S_0' with $m_0' \notin N_0 \cup N_0'$ and since A_1 is well-formed $(N_0 \cup N_0') \cap \mathrm{names}(\mathrm{skel}\, E_0') = \emptyset$, hence $(N \cup N') \cap \mathrm{names}(\mathrm{skel}\, E') = \emptyset$ as name capture is avoided when applying a realisation, proving well-formedness of functor signatures.

We have now proved well-formedness of $(\varphi A_1, \varphi A_2)$, using well-formedness of (A_1, A_2), that φA_2 covers φA_1 and that φA_2 is well-formed.

Consistency of type structures: Let (θ_1, CE_1) and (θ_2, CE_2) be arbitrary type-structures occuring in $(\varphi A_1, \varphi A_2)$. If they both occur in φA_2, consistency follows from consistency of φA_2. Consider the case where both (θ_1, CE_1) and (θ_2, CE_2) occur in φA_1, and assume $\theta_1 = \theta_2$. Then there exist (θ_1', CE_1') and (θ_2', CE_2') occuring in A_1 with $\mathrm{Dom}\, CE_1' = \mathrm{Dom}\, CE_1$

and $\mathrm{Dom}\, CE'_2 = \mathrm{Dom}\, CE_2$. Then as A_2 covers A_1, there exist (θ'_1, CE''_1) and (θ'_2, CE''_2) occuring in A_2, such that $\mathrm{Dom}\, CE''_1 = \mathrm{Dom}\, CE'_1$ or $\mathrm{Dom}\, CE'_1 = \emptyset$ and $\mathrm{Dom}\, CE''_2 = \mathrm{Dom}\, CE'_2$ or $\mathrm{Dom}\, CE'_2 = \emptyset$. Then there exist $\varphi(\theta'_1, CE''_1) = (\theta_1, \varphi CE''_1)$ and $\varphi(\theta'_2, CE''_2) = (\theta_2, \varphi CE''_2)$ occuring in φA_2 with $\mathrm{Dom}(\varphi CE''_1) = \mathrm{Dom}\, CE''_1$ and $\mathrm{Dom}(\varphi CE''_2) = \mathrm{Dom}\, CE''_2$. Then as φA_2 is consistent, $\mathrm{Dom}(\varphi CE''_1) = \mathrm{Dom}(\varphi CE''_2)$ or $\mathrm{Dom}(\varphi CE''_1) = \emptyset$ or $\mathrm{Dom}(\varphi CE''_2) = \emptyset$. Assume $\mathrm{Dom}(\varphi CE''_1) = \mathrm{Dom}(\varphi CE''_2)$. Then $\mathrm{Dom}(CE''_1) = \mathrm{Dom}(CE''_2)$. It is only interesting to consider the case where $\mathrm{Dom}\, CE''_1 = \mathrm{Dom}\, CE'_1$ and $\mathrm{Dom}\, CE''_2 = \mathrm{Dom}\, CE'_2$ (if $\mathrm{Dom}\, CE'_1 = \emptyset$ then as $\mathrm{Dom}\, CE'_1 = \mathrm{Dom}\, CE_1$ consistency is trivially satisfied, analogously if $\mathrm{Dom}\, CE'_2 = \emptyset$). But then we have $\mathrm{Dom}\, CE'_1 = \mathrm{Dom}\, CE'_2$, hence $\mathrm{Dom}\, CE_1 = \mathrm{Dom}\, CE_2$ proving type-structure consistency of φA_1. Now consider the case where (θ_1, CE_1) occurs in φA_1 and (θ_2, CE_2) occurs in φA_2. But then consistency immediately follows from φA_2 covering φA_1.

Consistency of structures: Let S_1 and S_2 occur free in $(\varphi A_1, \varphi A_2)$ and assume m of $S_1 = m$ of S_2, and that there exists a *longstrid* $\in \mathrm{Dom}\, S_1 \cap \mathrm{Dom}\, S_2$. Let $(m, E_1) = S_1$ and $(m, E_2) = S_2$. Consider the case where S_1 and S_2 both occur in φA_1. Then there exist (m'_1, E'_1) and (m'_2, E'_2) occuring in A_1 such that $\varphi(m'_1, E'_1) = (m, E_1)$ and $\varphi(m'_2, E'_2) = (m, E_2)$ and *longstrid* $\in \mathrm{Dom}\, E'_1 \cap \mathrm{Dom}\, E'_2$. Since A_2 covers A_1 there exist structures (m'_1, E''_1) and (m'_2, E''_2) occuring in A_2 such that *longstrid* $\in \mathrm{Dom}\, E''_1 \cap E''_2$. Thus *longstrid* $\in \mathrm{Dom}(\varphi E''_1) \cap \mathrm{Dom}(\varphi E''_2)$. As φA_2 is consistent m of $(\varphi E''_1)(\mathit{longstrid}) = m$ of $(\varphi E''_2)(\mathit{longstrid})$ and therefore m of $(\varphi E'_1)(\mathit{longstrid}) = m$ of $(\varphi E'_2)(\mathit{longstrid})$, as required. Consider the case where S_1 occurs in φA_1 and S_2 occurs in φA_2. Then there exist (m'_1, E'_1) occuring in A_1 with *longstrid* $\in \mathrm{Dom}\, E'_1$ and (m'_2, E'_2) occuring in A_2 with *longstrid* $\in \mathrm{Dom}\, E'_2$. As A_2 covers A_1, there exists a structure (m'_1, E''_2) occuring in A_2 such that *longstrid* $\in \mathrm{Dom}\, E''_2$. Hence $\varphi(m'_1, E''_2) = (m, \varphi E''_2)$ occurs in φA_2 and *longstrid* $\in \mathrm{Dom}(\varphi E''_2)$. Since φA_2 is consistent we have that m of $E_2(\mathit{longstrid}) = m$ of $(\varphi E''_2)(\mathit{longstrid})$ and as A_1, A_2 is consistent, we have m of $E'_1(\mathit{longstrid}) = m$ of $E''_2(\mathit{longstrid})$. Hence

$$\begin{aligned} m \text{ of } E_1(\mathit{longstrid}) &= \varphi(m \text{ of } E'_1(\mathit{longstrid})) \\ &= \varphi(m \text{ of } E''_2(\mathit{longstrid})) \\ &= m \text{ of } (\varphi E''_2)(\mathit{longstrid}) \\ &= m \text{ of } E_2(\mathit{longstrid}) \end{aligned}$$

as required. Now assume that there exists a *tycon* $\in \mathrm{Dom}\, S_1 \cap \mathrm{Dom}\, S_2$; by an argumentation as the above we get the required.

We have now proved consistency of $(\varphi A_1, \varphi A_2)$, using that A_2 covers A_1 and that φA_2 is consistent, and that (A_1, A_2) is consistent.

Cycle-free: $(\varphi A_2, \varphi A_1)$ is cycle-free because φA_2 is cycle-free and φA_2 covers φA_1.

Now we have proved admissibility of $(\varphi A_2, \varphi A_1)$. $\square$

Let $\mathcal{CR}$ denote the rules 4.17–4.21 together with rules 47–52 in the Definition and let $\mathcal{MR}$ denote the following rules 4.1–4.16, 4.22–4.27 together with $\mathcal{CR}$ (all the presented rules, except the one for signature bindings).

Now we can prove the fundamental realisation theorem — note that one is allowed to add components to flexible structures (expressed by $O \xrightarrow{\varphi} O'$ in the theorem) as discussed in Section 4.11.

Theorem 5.2.1 (Realisation) *If $P \vdash phrase \Rightarrow Q$ from $\mathcal{CR}$ and $\varphi(P, Q)$ is admissible then $\varphi P \vdash phrase \Rightarrow \varphi Q$ from $\mathcal{CR}$. Moreover, if $O \vdash phrase \Rightarrow Q$ from $\mathcal{MR}$ and $O \xrightarrow{\varphi} O'$ then $O' \vdash phrase \Rightarrow \varphi Q$ from $\mathcal{MR}$.*

Proof By induction on the depth of the inference tree. The first part of the theorem concerning the rules in $\mathcal{CR}$ is proved as in the Commentary [11, Page 99].[2] Now to the second part of the theorem. The cases for rules 4.1, 4.17–4.16, 4.22–4.26 are all straightforward using the notes in the Commentary, Page 99–100. The cases for the four remaining rules follow below. These are taken from [16] (they carry over smoothly).

Rule 4.2, *sigexp* $\equiv$ *sigid*

Let $(A, B) = O$ and $(A', B') = O'$. Assume $O \xrightarrow{\varphi} O'$ and $O \vdash sigid \Rightarrow S$. By rule 4.2 we have $B(sigid) \geq S$. Thus $(\varphi B)(sigid) \geq \varphi S$, i.e. $B'(sigid) \geq \varphi S$, by Lemma 5.2.1. Moreover, since $S \sqsubseteq A$ we have $\varphi S \sqsubseteq \varphi A \sqsubseteq A'$, by Lemma 5.2.3. Thus $O' \vdash sigexp \Rightarrow \varphi S$, as required.

[2] For the pedantic reader: In the proof in the Commentary, the note to the case for rule 86 should be $\text{tyvars}(\varphi ty) \subseteq \text{tyvars}(ty)$ instead of $\text{tyvars}(\varphi\tau) \subseteq \text{tyvars}(\tau)$, c.f. the correction to rule 86 in Appendix D of the Commentary.

Rule 4.3, principal signatures

Let $(A, B) = O$ and $(A', B') = O'$. By rule 4.3, $P = (N)S$, for some N and S, and also there exists an A_1 such that $A \trianglelefteq A_1$ and $N \cap \text{names}\, A = \emptyset$, $(N)S$ is principal for *sigexp* in O and

$$A_1, B \vdash \mathit{sigexp} \Rightarrow S \tag{5.1}$$

Since $A \trianglelefteq A_1$ and $O \xrightarrow{\varphi} O'$ there exists a φ_1 and an A_1' such that $A' \trianglelefteq A_1'$, φ_1 is injective on N_1 and $\varphi_1(N_1) = N_1'$, $\varphi_1(n) = \varphi(n)$, for all $n \in \text{names}\, A$, and

$$(A_1, B) \xrightarrow{\varphi_1} (A_1', B') \tag{5.2}$$

where $N_1 = \text{names}\, A_1 \setminus \text{names}\, A$ and $N_1' = \text{names}\, A_1' \setminus \text{names}\, A'$. By induction on (5.1) and (5.2) we have

$$A_1', B' \vdash \mathit{sigexp} \Rightarrow \varphi_1 S \tag{5.3}$$

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 $O \vdash \mathit{sigexp} \Rightarrow (N)S$. Thus $\varphi((N)S) = (N')(\varphi_1 S)$. Is this signature principal for *sigexp* in O'? Certainly $A' \trianglelefteq A_1'$ and $A_1', B' \vdash \mathit{sigexp} \Rightarrow \varphi_1 S$, 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 $O'' \vdash \mathit{sigexp} \Rightarrow S'$. 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 4.3 to (5.3) and get $O' \triangleright \mathit{sigexp} \Rightarrow (N')(\varphi_1 S)$, i.e. $O' \triangleright \mathit{sigexp} \Rightarrow \varphi((N)S)$. Then, since $A \sqsupseteq (N)S$ and $O \xrightarrow{\varphi} O'$ we have $A' \sqsupseteq \varphi((N)S)$, by Lemma 5.2.3, so $O' \vdash \mathit{sigexp} \Rightarrow \varphi((N)S)$, as required.

Rule 4.27, *funsigexp* $\equiv$ (*strid* : *sigexp*$_1$) : *sigexp*$_2$

Assume $O \xrightarrow{\varphi} O'$ and $O \vdash \mathit{funsigexp} \Rightarrow \Phi$. Then *funsigexp* is of the form (*strid* : *sigexp*$_1$) : *sigexp*$_2$ and Φ is of the form $(N_1)(S_1, \Sigma_2)$, where $N_1 \cap \text{names}(O) = \emptyset$. Let $(A, B) = O$ and $(A', B') = O'$. By rule 4.27 we have

$$A, B \vdash \mathit{sigexp}_1 \Rightarrow (N_1)S_1 \tag{5.4}$$

$$(S_1, A), B + N_1 + \{\mathit{strid} \mapsto S_1\} \vdash \mathit{sigexp}_2 \Rightarrow \Sigma_2 \tag{5.5}$$

Without loss of generality we can assume that $N_1 \cap \text{names}\, O' = \emptyset$ and that $\varphi n = n$, for all $n \in N_1$. Then by induction on (5.4) we have $A', B' \vdash \mathit{sigexp}_1 \Rightarrow \varphi((N_1)S_1)$, i.e.

$$A', B' \vdash \mathit{sigexp}_1 \Rightarrow (N_1)(\varphi S_1) \tag{5.6}$$

We have $(S_1, A), B + N_1 + \{strid \mapsto S_1\} \xrightarrow{\varphi} (\varphi S_1, A'), B' + N_1 + \{strid \mapsto \varphi S_1\}$, so by induction using (5.5) we have

$$(\varphi S_1, A'), B' + N_1 + \{strid \mapsto \varphi S_1\} \vdash sigexp_2 \Rightarrow \varphi\Sigma_2 \tag{5.7}$$

Since N_1 is chosen disjoint from names(O, O') and $A \sqsupseteq (N_1)S_1$ and $(S_1, A) \sqsupseteq \Sigma_2$ (by (5.4) and (5.5), respectively), we have that $\varphi\Phi = (N_1)(\varphi S_1, \varphi\Sigma_2)$. Finally, from $A \sqsupseteq \Phi$ and $O \xrightarrow{\varphi} O'$ we get $A' \sqsupseteq \varphi\Phi$, by Lemma 5.2.3. We can then combine (5.6) and (5.7) to the desired $O' \vdash funsigexp \Rightarrow \varphi\Phi$. □

Chapter 6

Principal Signatures

In this chapter we state and prove the existence of principal signatures. We start by defining the Below operation described informally in Chapter 3 and then prove a couple of lemmas expressing fundamental properties of this operation. In the next section we prove a series of lemmas concerned with principality and we prove the existence of principal elaborations which is the main theorem used to prove the existence of principal signatures.

6.1 Names below names

Given an assembly in which some components possibly have been added to some structures or type structures (whose names are all members of a name set N) the following definition is used to get a hold on the names of those added components. It is used to "implement" the Below operation, see Section 3.3. The definition, and also the following, are changed compared to the definitions in [16] as we treat the full language.

Definition 6.1.1 (Structure names below) *Let A be a type-explicit assembly and let N be a name set. The names below N in A, written* $\mathrm{below}(A, N)$*, is the least name set N', satisfying*

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

Definition 6.1.2 (Type structures and structures below) *Let A be a type-explicit assembly and let N be a name set. The type structures and structures below N in A is* *defined by*

$$\begin{aligned}\mathrm{Below}(A,N) &= \{\mathrm{skel}(m,E) \mid (m,E) \textit{ occurs free in } A \textit{ and } m \in \mathrm{below}(A,N)\}\\ &\cup \{\mathrm{skel}(\theta,CE) \mid (\theta,CE) \textit{ occurs free in } A \textit{ and } \theta \in \mathrm{below}(A,N)\}\end{aligned}$$

Notice how the skel function is used to exempt functor components and variable and exception environments and how the skel function makes the range of constructor environments trivial as discussed in Section 3.3

When A' is a semantic object, we write $\mathrm{below}(A, A')$ as an abbreviation of $\mathrm{below}(A, \mathrm{names}\, A')$ and we write $\mathrm{Below}(A, A')$ as an abbreviation of $\mathrm{Below}(A, \mathrm{names}\, A')$. Informally, $\mathrm{Below}(A, A')$ is the set of structures and type structures in A which "stem from" A', i.e. which are (larger) views of structures and type structures occuring in A'. We shall identify the set $\mathrm{Below}(A, M)$ with any assembly $[S_1, \ldots, S_n, (\theta_1, CE_1), \ldots, (\theta_m, CE_m)]$, where $\{S_1, \ldots, S_n\, (\theta_1, CE_1), \ldots, (\theta_m, CE_m)\} = \mathrm{Below}(A, M)$. The following lemma is analogous to a lemma in [16], but the proof is, of course, not the same as the definitions we use are different since we consider the full language.

Lemma 6.1.1 *Let A be a type-explicit assembly and let N be a name set. Then* $\mathrm{below}(A, N) = \mathrm{names}(\mathrm{Below}(A, N))$.

Proof We first show $\mathrm{below}(A, N) \subseteq \mathrm{names}(\mathrm{Below}(A, N))$. Take $m \in \mathrm{below}(A, N)$. Then there exists an E such that (m, E) occurs free in A. Since $m \in \mathrm{below}(A, N)$ we have $\mathrm{skel}(m, E) \in \mathrm{Below}(A, N)$. Thus $m \in \mathrm{names}(\mathrm{Below}(A, N))$. Now take $t \in \mathrm{below}(A, N)$. Then there exists a CE such that (t, CE) occurs free in A as A is type-explicit. Since $t \in \mathrm{below}(A, N)$ we have $\mathrm{skel}(t, CE) \in \mathrm{Below}(A, N)$. Thus $t \in \mathrm{names}(\mathrm{Below}(A, N))$. We then show $\mathrm{below}(A, N) \supseteq \mathrm{names}(\mathrm{Below}(A, N))$. Take a name $m \in \mathrm{names}(\mathrm{Below}(A, N))$. Then m occurs free in some $(m', E') \in \mathrm{Below}(A, N)$. By the definition of Below, there exits an E such that (m', E) occurs free in A and $m' \in \mathrm{below}(A, N)$ and $\mathrm{skel}(m', E) = (m', E')$. Then, by the definition of below, we have $\mathrm{names}(m', E') \subseteq \mathrm{below}(A, N)$ so in particular, $m \in \mathrm{below}(A, N)$. Now take $t \in \mathrm{names}(\mathrm{Below}(A, N))$. Then, by the definition of skel and Below, either there exists a CE such that (t, CE) occurs free in A and $t \in \mathrm{below}(A, N)$ as required, or there exists a (m, E) such that (m, E) occurs free in A and $m \in \mathrm{below}(A, N)$ and $t \in \mathrm{names}(\mathrm{skel}(m, E))$ but then, by definition of below, we have $\mathrm{names}(\mathrm{skel}(m, E)) \subseteq \mathrm{below}(A, N)$ so in particular, $t \in \mathrm{below}(A, N)$ as required. □

Informally, the following lemma simply expresses that the Below operation does not return a larger set of structures than expected. The lemma is analogous to a lemma in [16], but our proof is a bit different due to different definitions and written out in more detail.

Lemma 6.1.2 *Let A be an admissible type-explicit assembly 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 $\mathrm{names}(\mathrm{Below}(A, N))$. Surely $\mathrm{names}(\mathrm{Below}(A, N)) \cap \mathrm{names}\, A \subseteq \mathrm{names}(\mathrm{Below}(A, N))$. Now assume (m, E) occurs free in A and $m \in \mathrm{names}(\mathrm{Below}(A, N))$ and let id be a type constructor or a functor or structure identifier such that $id \in \mathrm{Dom}\, E$. Then by Lemma 6.1.1, $m \in \mathrm{below}(A, N)$ and by definition of Below, $\mathrm{skel}(m, E)$ occurs free in $\mathrm{Below}(A, N)$ and, by definition of skel, $id \in \mathrm{Dom}(\mathrm{skel}\, E)$. Let (θ, CE) occur free in A and assume $\theta \in \mathrm{names}(\mathrm{Below}(A, N))$. By Lemma 6.1.1, $\theta \in \mathrm{below}(A, N)$ so by definition of Below, $\mathrm{skel}(\theta, CE) = (\theta, \mathrm{skel}\, CE)$ occurs free in $\mathrm{Below}(A, N)$ and as $\mathrm{Dom}\, CE = \mathrm{Dom}(\mathrm{skel}\, CE)$ we have the required. □

The following lemma and its proof are almost identical to a lemma and proof in [16].

Lemma 6.1.3 *Let B_1 be a basis, and let A_1 and A'_1 be type-explicit assemblies. 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 5.2.3. Since $\varphi A_1 \sqsubseteq A_2$, we then have $\varphi A'_1 \sqsubseteq A_2$. Thus

$$A'_2 = \mathrm{Below}(A_2, \varphi A'_1) \sqsupseteq \varphi A'_1 \tag{6.1}$$

Since by assumption $A'_1 \sqsupseteq B_1$ and A'_1 is type-explicit, we have $(A'_1, B_1) \in \mathrm{Obj}$. To see that $(A'_2, B_2) \in \mathrm{Obj}$, note that A'_2 is type-explicit (as A_2 is type-explicit and by definition of Below) and that $A'_1 \sqsupseteq B_1$ implies $\varphi A'_1 \sqsupseteq \varphi B_1 = B_2$, by Lemma 5.2.3; thus $A'_2 \sqsupseteq B_2$, by (6.1). Finally, φ is a morphism from (A'_1, B_1) to (A'_2, B_2), for $\varphi A'_1 \sqsubseteq A'_2$ by (6.1) and moreover, A'_1 covers A'_2 on N of B_1, because A_1 covers A_2 on N of B_1 and A_1 is a *conservative* cover of A'_1, and $\mathrm{names}(A'_1) \supseteq N$ of B_1. □

In [16] Tofte gives a characterisation of his definition of below which often is useful in proofs. We here give an analogous characterisation of our below — specifically we will use it in the proof of Lemma 6.2.8. Recall that Name is the set of all names. Let $\mathcal{P}(\mathrm{Name})$ denote the set of subsets of Name. Given A and N, let $\mathcal{F}_{A,N} : \mathcal{P}(\mathrm{Name}) \to \mathcal{P}(\mathrm{Name})$ be defined by

$$\begin{aligned}\mathcal{F}_{A,N}(N') = \{n \in \mathrm{names}\, A \mid\ & n \in N \text{ or there exists an } (m', E') \text{ which} \\ & \text{occurs free in } A \text{ and satisfies that } m' \in N' \\ & \text{and } n \in \mathrm{names}(\mathrm{skel}\, E') \qquad\qquad \}\end{aligned}$$

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

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

6.2 Closure And Principality

The following two lemmas are easily proved (the second is proved using the first.)

Lemma 6.2.1 *Let $A \sim A'$. Then $(N)S$ is principal for sigexp in (A, B) if and only if $(N)S$ is principal for sigexp in (A', B).*

Lemma 6.2.2 *Let $A \sim A'$. Then $A, B \vdash$ phrase $\Rightarrow Q$ from $\mathcal{MR}$ if and only if $A', B \vdash$ phrase $\Rightarrow Q$ from $\mathcal{MR}$.*

Definition 6.2.1 (Closure) *For all structures S and name sets N, we define closure of S with respect to N, written $\text{Clos}_N S$, to be the signature $(N')S$, where $N' = \text{names}\, S \setminus N$. For every semantic object A, $\text{Clos}_A S$ means $\text{Clos}_{\text{names}\, A} S$.*

The following lemma is analogous to a lemma in [16], and the proof is almost the same.

Lemma 6.2.3 (Closure and Well-formedness, Version 1) *For all name sets N, type-explicit 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 $\text{names}(S) \setminus \text{names}(\text{Below}(A, N))$. By Lemma 6.1.1 we then have that $N' = \text{names}(S) \setminus \text{below}(A, N)$. 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 have by definition of Below that $\text{skel}(m, E)$ occurs free in $\text{Below}(A, N)$ and we therefore have $\text{names}(\text{skel}\, E) \subseteq \text{names}(\text{Below}(A, N))$. Hence $\text{names}(\text{skel}\, E) \cap N' = \emptyset$, as required. □

The following lemma is in itself perhaps not the most interesting in the world but the proof might be worth reading as it precisely expresses our intuitions of how the operations Below and skel and the concepts of consistency, type explication, and cover play together. If the skel function did not exempt functor components we would need a stronger notion of consistency (as in [15]) to prove the lemma.

Lemma 6.2.4 *Let A and A' be type-explicit assemblies. If $A \trianglelefteq A'$ then* $\mathrm{names}\,A = \mathrm{names}(\mathrm{Below}(A', \mathrm{names}\,A))$.

In the proof we make use of the concept *path* defined as follows.

Definition 6.2.2 (Path) *A* structure path *is a finite string over the alphabet* StrId. *We use the notation $strid_1.\cdots.strid_k$, $k \geq 0$, for structure paths. A* type constructor path *is a finite string over the alphabet* StrId $\cup$ TyCon *of the form $strid_1.\cdots.strid_k.tycon$, $k \geq 0$. For a structure (m, E) and a name n, a* path from m to n *is a path $strid_1.\cdots.strid_k.id$, $k \geq 0$, where $id \in$* StrId $\cup$ TyCon, *for which n* of *$E(strid_1.\cdots.strid_k.id) = n$.*

Proof First note that, by Lemma 6.1.1, $\mathrm{names}(\mathrm{Below}(A', \mathrm{names}\,A)) = \mathrm{below}(A', \mathrm{names}\,A)$. We first show $\mathrm{names}\,A \subseteq \mathrm{names}(\mathrm{Below}(A', \mathrm{names}\,A))$. Let $t \in \mathrm{names}\,A$. As A is type-explicit, there exists a CE such that (t, CE) occurs free in A. As $A \trianglelefteq A'$ we get by definition of cover that there exists a CE' such that (t, CE') occurs free in A', that is $t \in \mathrm{names}\,A'$ so by definition of below we have that $t \in \mathrm{below}(A', namesA)$. Let $m \in \mathrm{names}\,A$. Then there exists an E such that (m, E) occurs free in A, so by definition of cover there exists an E' such that (m, E') occurs free in A'. Thus $m \in \mathrm{names}\,A'$, so by definition of below, we have $m \in \mathrm{below}(A', namesA)$ as required. We then show $\mathrm{names}\,A \supseteq \mathrm{names}(\mathrm{Below}(A', \mathrm{names}\,A))$. Using the alternative characterisation of below we must prove that

$$\bigcup_{i \geq 0} \mathcal{F}^{i}_{A', \mathrm{names}\,A}(\emptyset) \subseteq \mathrm{names}\,A$$

This is done by induction on i.
Base Case, $i = 0$. Holds vacously as $\mathcal{F}^{0}_{A', \mathrm{names}\,A}(\emptyset) = \emptyset$.
Inductive Step, $i > 0$. We assume it holds for i and then show that it holds for $i + 1$. Let $n \in \mathcal{F}^{(i+1)}_{A', \mathrm{names}\,A}(\emptyset)$. Then either $n \in \mathrm{names}\,A$ as required or there exists a (m', E') which occurs free in A' and satisfies that $m' \in \mathcal{F}^{i}_{A', \mathrm{names}\,A}(\emptyset)$ and $n \in \mathrm{names}(\mathrm{skel}\,E')$. Due to the application of the skel function there is a path from m' to n in A'. By induction we have that $m' \in \mathrm{names}\,A$. We then show by induction on the length k of the path from m' to n in A' that $n \in \mathrm{names}\,A$.
Base Case, $k = 0$. Then there exists an $id \in \mathrm{StrId} \cup \mathrm{TyCon}$ such that n of $E'(id) = n$. By conservative cover there exists an E'' such that (m', E'') occurs free in A and $id \in \mathrm{Dom}\,E''$. Then by consistency n of $E''(id) = n$ of $E'(id)$, so $n \in \mathrm{names}\,A$ as required.
Inductive Step, $k > 0$. We assume it holds for length k and show that it holds for length $k + 1$. The argumentation is analogous to the argumentation for the base case. $\square$

The following three lemmas and proofs are almost identical to lemmas and proofs in [16].

Lemma 6.2.5 (Closure and Well-formedness, Version 2) *For all type-explicit assemblies A and A' and all structures S, if $A \trianglelefteq A'$ and $A' \sqsupseteq S$ then $\mathrm{Clos}_A S$ is a well-formed signature.*

Proof Follows directly from Lemma 6.2.3 and Lemma 6.2.4. □

Lemma 6.2.6 (Closure and Principality) *Let $O = (A, B)$ be an object in K. For all name sets N and structures S, if $N \cap \mathrm{names}\, O = N \setminus \mathrm{names}(S) = \emptyset$ and $(N)S$ is principal for sigexp in O then $(N)S = \mathrm{Clos}_A S$.*

Proof Since $N \setminus \mathrm{names}(S) = \emptyset$ we know that $N \subseteq \mathrm{names}\, S$. Since $N \cap \mathrm{names}\, O = \emptyset$, we know that $N \cap \mathrm{names}\, A = \emptyset$. To show $(N)S = \mathrm{Clos}_A S$, it just remains to show that $\mathrm{names}((N)S) \subseteq \mathrm{names}\, A$. 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

$$\text{For all } O', \varphi \text{ and } S', \text{ if } O \xrightarrow{\varphi} O' \text{ and } O' \vdash sigexp \Rightarrow S' \text{ then } \varphi((N)S) \geq S' \tag{6.2}$$

In particular, for $\varphi = \mathrm{Id}$, (6.2) states that

$$\text{For all } A'' \text{ and } S', \text{ if } A \sqsubseteq A'' \text{ and } A'', B \vdash sigexp \Rightarrow S' \text{ then } (N)S \geq S' \tag{6.3}$$

Let φ be an injective realisation which maps names in the set $\mathrm{names}(A') \setminus \mathrm{names}(A)$ to distinct fresh names and is the identity on all other names. Since $A \trianglelefteq A'$ we have that $A \trianglelefteq \varphi A'$ and since $A \sqsupseteq B$, we 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 5.2.1. Thus by (6.3) we have $(N)S \geq \varphi S$. But this implies $\mathrm{names}((N)S) \subseteq \mathrm{names}\, A$, as required. □

Lemma 6.2.7 (Realisation and Principality) *Let $O_i = (A_i, B_i)$ be objects in K, for $i = 1, 2$, let φ be a realisation 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 $N_i \cap \mathrm{names}\, A_i = N_i \setminus \mathrm{names}\, S_i = \emptyset$ 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:

$$\varphi \text{ is injective on } N_1 \text{ and } \varphi N_1 \subseteq N_2 \tag{6.4}$$

without capture of names:

$$N_2 \cap \varphi(\mathrm{names}((N_1)S_1)) = \emptyset \tag{6.5}$$

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 realisation which satisfies $\varphi'(n) = \varphi(n)$, for all $n \in \mathrm{names}(A_1)$ but in addition maps all names in the set $\mathrm{names}(A'_1) \setminus \mathrm{names}(A_1)$ to distinct fresh names. By Lemma 6.2.6 we $(N_i)S_i = \mathrm{Clos}_{A_i} S_i$, $i = 1, 2$. In particular, $N_1 \subseteq \mathrm{names}\, A'_1 \setminus \mathrm{names}\, A_1$, so

$$\varphi' \text{ maps the names in } N_1 \text{ to distinct fresh names} \tag{6.6}$$

Since $A_1 \trianglelefteq A'_1$ and $(A_1, B_1) \xrightarrow{\varphi} (A_2, B_2)$ we have $\varphi A_1 \sqsubseteq A_2 \sqsubseteq A'_2$ 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 5.2.1. 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 (6.6), gives (6.4). As for (6.5), we have $\mathrm{names}((N_1)S_1) \subseteq \mathrm{names}(A_1)$. Since $(N_2)(\varphi S_1) = \mathrm{Clos}_{A_2}(\varphi S_1)$ and $\varphi A_1 \sqsubseteq A_2$ we therefore have (6.5), as desired. □

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 $\psi_1 = \psi_2$. The following lemma is the one we referred to in Section 3.3; it is taken from [16], but our proof is different as we consider the full language and written out in more detail (for instance, we present the induction.)

Lemma 6.2.8 *Let $(A_1, B_1) = O_1$ and $(A_2, B_2) = O_2$ and assume $O_1 \xrightarrow{\varphi} O_2$.*
Then $O_1 \xrightarrow{\varphi} (\mathrm{Below}(A_2, \varphi A_1), B_2)$ is an epimorphism.

Proof Let $A'_2 = \mathrm{Below}(A_2, \varphi A_1)$ 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 realisations such that

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

commutes. We wish to prove that $\psi_1 = \psi_2$. By the definition of A'_2 this amounts to proving that

$$\begin{aligned}
&\forall n \in \mathrm{names}(\mathrm{Below}(A_2, \varphi A_1)).\psi_1(n) = \psi_2(n) \iff \\
&\forall i \forall n \in \mathrm{names}(\{\, \mathrm{skel}(m, E) \mid \ (m, E) \text{ which occurs free in } A_2 \text{ and } m \in \mathcal{F}^i_{(A_2, \varphi A_1)}(\emptyset) \,\} \\
&\qquad\qquad \cup \{\, \mathrm{skel}(\theta, CE) \mid \ (\theta, CE) \text{ occurs free in } A_2 \text{ and } \theta \in \mathcal{F}^i_{(A_2, \varphi A_1)}(\emptyset) \,\}). \\
&\quad \psi_1(n) = \psi_2(n)
\end{aligned}$$

This we prove by induction on i. It holds vacously for the base case ($i = 0$), as $\mathcal{F}^0_{(A_2,\varphi A_1)}(\emptyset) = \emptyset$. Inductive case: We assume it holds for i, and then show that it holds for $i+1$. There are two times two cases to consider, c.f. the definition of $\mathcal{F}$.

Case 1. Assume $m \in \mathrm{names}(\varphi A_1)$ and (m, E) occurs free in A_2 for some E. Then $m = \varphi m'$ for some m'. As $\psi_1 \circ \varphi = \psi_2 \circ \varphi$, we have $\psi_1 m = \psi_2 m$. Let $A_3 = A$ of O_3. Since $\psi_1 A_2' \sqsubseteq A_3$ and $\psi_2 A_2' \sqsubseteq A_3$ and A_3 is consistent, we have $\forall n \in \mathrm{names}(\mathrm{skel}(m, E)).\psi_1(n) = \psi_2(n)$.

Case 2. Assume (m, E) occurs free in A_2 and that there exists a (m', E') occurring free in A_2 such that $m' \in \mathcal{F}^i_{(A_2,\varphi A_1)}(\emptyset)$ and $m \in \mathrm{names}(\mathrm{skel}\, E')$. Then by induction, $\psi_1(n) = \psi_2(n)$, hence by the same argumentation as above, $\forall n \in \mathrm{names}(\mathrm{skel}(m, E)).\psi_1(n) = \psi_2(n)$.

Case 3. Assume (θ, CE) occurs free in A_2 and $\theta \in \mathrm{names}(\varphi A_1)$. Then $\theta = \varphi t'$ for some t'. As $\psi_1 \circ \varphi = \psi_2 \circ \varphi$ we have $\psi_1 \theta = \psi_2 \theta$, and as $\mathrm{names}(\mathrm{skel}(\theta, CE)) = \{\theta\}$ we have the required.

Case 4. Assume (θ, CE) occurs free in A_2 and that there exists a (m', E') which occurs free in A_2 and $m' \in \mathcal{F}^i_{(A_2,\varphi A_1)}(\emptyset)$ and $\theta \in \mathrm{names}(\mathrm{skel}\, E')$. Then by induction, $\forall n \in \mathrm{names}(\mathrm{skel}(m', E')).\psi_1(n) = \psi_2(n)$, so in particular $\psi_1 \theta = \psi_2 \theta$. Thus as $\mathrm{names}(\mathrm{skel}(\theta, CE)) = \{\theta\}$ we have the required. □

The following lemma and proof is taken from [16].

Lemma 6.2.9 (The Principality Theorem gives Principal Signatures) *Assume $O \xrightarrow{\varphi^*} O^*$ and $O^* \vdash sigexp \Rightarrow S^*$. Further, assume that for all O', S' and φ, if $O \xrightarrow{\varphi} O'$ and $O' \vdash sigexp \Rightarrow S'$ then there exists a ψ such that the diagram*

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.7}$$

commutes and $\psi S^* = S'$. Let $(A, B) = O$, $(A^*, B^*) = O^*$, $A_0^* = \mathrm{Below}(A^*, \varphi^* A)$ and $\Sigma^* = \mathrm{Clos}_{A_0^*} S^*$. Then Σ^* is principal for sigexp in (A_0^*, B^*).

Proof Following the definition of principal signature, it suffices to prove

$$(A_0^*, B^*) \in \mathrm{Obj} \tag{6.8}$$

$$A_0^* \trianglelefteq A^* \quad \text{and} \quad A^*, B^* \vdash sigexp \Rightarrow S^* \tag{6.9}$$

$$\text{For all } \varphi,\, O' \text{ and } S', \text{ if } (A_0^*, B^*) \xrightarrow{\varphi} O' \text{ and } O' \vdash \textit{sigexp} \Rightarrow S' \text{ then } \varphi(\Sigma^*) \geq S' \tag{6.10}$$

Now (6.8) follows from Lemma 6.1.3. Also, (6.9) follows from Lemma 6.1.2 and the assumption $O^* \vdash \textit{sigexp} \Rightarrow S^*$. Let us now prove (6.10). Let φ, O' and S' be such that $(A_0^*, B^*) \xrightarrow{\varphi} O'$ and $O' \vdash \textit{sigexp} \Rightarrow S'$. Since $O \xrightarrow{\varphi^*} O^*$ we have $O \xrightarrow{\varphi^*} (A_0^*, B^*)$ by Lemma 6.1.3. Thus the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi \circ \varphi^*"] & \\ (A_0^*, B^*) \arrow[rr, "\varphi"] & & O' \end{tikzcd} \tag{6.11}$$

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

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi \circ \varphi^*"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.12}$$

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

$$\varphi(m) = \psi(m), \text{ for all } m \in \mathrm{names}(A_0^*) \tag{6.13}$$

Since (6.12) commutes, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi \circ \varphi^*"] & \\ (A_0^*, B^*) \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.14}$$

commutes. By Lemma 6.2.8 we have that $O \xrightarrow{\varphi^*} (A_0^*, B^*)$ is an epimorphism. But then, since (6.11) and (6.14) commute, we have (6.13), as required. $\square$

The following lemma is taken from [16], but our proof is extended as we consider the full language.

Lemma 6.2.10 *Let* $O_i = (A_i, B_i)$, $i = 1..3$. *Let* $A_1' \trianglelefteq A_1$ *and* $A_3' \trianglelefteq A_3$ *and* $\varphi A_1' \sqsubseteq A_3'$. *Let* $N = \mathrm{names}\,A_1 \setminus \mathrm{names}\,A_1'$ *and assume that we also have* $N = \mathrm{names}\,A_3 \setminus \mathrm{names}\,A_3'$

and $N \subseteq N$ of B_1. *Further, assume the diagram*

$$\begin{tikzcd} & O_1 \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O_2 \arrow[rr, "\psi"] & & O_3 \end{tikzcd} \tag{6.15}$$

commutes and $A_2 \sqsubseteq \mathrm{Below}(A_2, \varphi^* A_1)$. *Let* $A_2' = \mathrm{Below}(A_2, \varphi^* A_1')$. *Then* $A_2 \sim (A_2', \varphi^* A_1)$ *and* $N \cap \mathrm{names}\, A_2' = \emptyset$.

Proof We have $(A_2', \varphi^* A_1) \sqsubseteq A_2$ by the definition of A_2' and the fact that $O_1 \xrightarrow{\varphi^*} O_2$ and Lemma 6.1.2. Let us show the converse $A_2 \sqsubseteq (A_2', \varphi^* A_1)$ (It is not obvious that this holds because in $(A_2', \varphi^* A_1)$ we only have that the objects that stems from A_1' are covered, c.f. definition of A_2'. For the objects that stems from A_1 and not from A_1', we are not assured that components, not covered by $\varphi^* A_1$ have not been added to them in A_2. So what we need to prove, is that the added components for the structures in A_2 stemming from $A_1 \setminus A_1'$ are covered by $(A_2', \varphi^* A_1)$.) Since by assumption $A_2 \sqsubseteq \mathrm{Below}(A_2, \varphi^* A_1)$ it will suffice to prove

$$\mathrm{Below}(A_2, \varphi^* A_1) \sqsubseteq (A_2', \varphi^* A_1) \tag{6.16}$$

We prove this by first considering the objects in $\mathrm{Below}(A_2, \varphi^* A_1)$ that stem from type names and then secondly considering the objects that stem from structure names.

Case 1. We show that $\mathrm{Below}(A_2, \varphi^*(\mathrm{tynames}\, A_1)) \sqsubseteq (A_2', \varphi^* A_1)$.

$\mathrm{Below}(A_2, \varphi^*(\mathrm{tynames}\, A_1))$ contains only type structures so let (θ, CE) be a type structure occurring free in $\mathrm{Below}(A_2, \varphi^*(\mathrm{tynames}\, A_1))$. Then there exists (θ, CE_2) occurring free in A_2 such that $\mathrm{skel}(\theta, CE_2) = (\theta, CE)$ and $\theta \in \varphi^*(\mathrm{tynames}\, A_1)$. Thus there exists $t \in \mathrm{tynames}\, A$ such that $\varphi^* t = \theta$.

Case 1.1. Assume $t \in N$. Then as φ^* is fixed on N, $t = \theta$. And as A_1 covers A_2 on N there exists (t, CE_1) occurring free in A_1 such that $\mathrm{Dom}\, CE_1 = \mathrm{Dom}(\mathrm{skel}\, CE_2) = \mathrm{Dom}\, CE$ or $\mathrm{Dom}\, CE = \emptyset$. But then $\varphi^*(t, CE_1) = (t, \varphi^* CE_1)$ occurs free in $\varphi^* A_1$ and as $\mathrm{Dom}(\varphi^* CE_1) = \mathrm{Dom}(CE_1)$ we have the required.

Case 1.2. Assume $t \notin N$. Then $t \in \mathrm{names}\, A_1'$, and we have the required by definition of A_2'.

Case 2. We show that $\mathrm{Below}(A_2, \varphi^*(\mathrm{strnames}\, A_1)) \sqsubseteq (A_2', \varphi^* A_1)$. Consider the sequence $M_0 \subseteq M_1 \subseteq M_2 \subseteq \cdots$ of structure name sets defined by letting $M_0 = \emptyset$ and, for $i \geq 0$,

$$M_{i+1} = \{m \in \mathrm{strnames}\, A_1 \mid \forall E. \text{if } (m, E) \text{ occurs free in } A_1, \text{ then } \mathrm{strnames}(\mathrm{skel}\, E) \subseteq M_i\}$$

Note that $M_{k+1} = M_k = \mathrm{strnames}\, A_1$, for some k, since A_1 is finite and cycle-free. Thus we can prove what we want by proving by induction on i that $\mathrm{Below}(A_2, \varphi^* M_i) \sqsubseteq (A_2', \varphi^* A_1)$.
Base Case, $i = 0$. Here $\mathrm{Below}(A_2, \emptyset)$ is the empty assembly which trivially is covered by $(A_2', \varphi^* A_1)$.
Inductive Step, $i > 0$. Assume $\mathrm{Below}(A_2, \varphi^* M_{i-1}) \sqsubseteq (A_2', \varphi^* A_1)$. We wish to prove that $\mathrm{Below}(A_2, \varphi^* M_i) \sqsubseteq (A_2', \varphi^* A_1)$. As we already have $\mathrm{Below}(A_2, \varphi^* M_{i-1}) \sqsubseteq (A_2', \varphi^* A_1)$, it will suffice to prove that for all (m, E) occurring free in A_2 with $m \in \mathrm{below}(A_2, \varphi^* M_i) \setminus \mathrm{below}(A_2, \varphi^* M_{i-1})$ that $(m, \mathrm{skel}\, E) \sqsubseteq (A_2', \varphi^* A_1)$. So let (m, E) be such a structure. As $m \in \mathrm{below}(A_2, \varphi^* M_i) \setminus \mathrm{below}(A_2, \varphi^* M_{i-1})$ there exists an $m_i \in M_i \setminus M_{i-1}$ such that $m \in \mathrm{below}(A_2, \varphi^* m_i)$. (Intuitively m_i is either equal to m or occurs above m.) There are two cases to consider:
Case 1, $m_i \in M \text{ of } N$ (remember $N = \mathrm{names}\, A_1 \setminus \mathrm{names}\, A_1'$). As φ^* is fixed on N (as $N \subseteq N \text{ of } B_1$ and φ^* is fixed on $N \text{ of } B_1$, c.f. definition of morphism in K), we have $\varphi^* m_i = m_i$ and hence $m \in \mathrm{below}(A_2, m_i)$. However since $m \notin \mathrm{below}(A_2, \varphi^* M_{i-1})$ and A_1 covers A_2 on N (as $N \subseteq N \text{ of } B_1$), we must have $m = m_i$.
Let *strid* be a structure identifier in $\mathrm{Dom}\, E$. Since A_1 covers A_2 on N there exists an environment E_1 such that (m, E_1) occurs free in A_1 and $\mathit{strid} \in \mathrm{Dom}\, E_1$. Thus $\varphi^*(m, E_1) \sqsubseteq \varphi^* A_1$. But then $\varphi^*(m \text{ of } E_1(\mathit{strid})) = m \text{ of } E(\mathit{strid})$, since A_2 is consistent and $\varphi^* A_1 \sqsubseteq A_2$ and $\varphi^* m = m$. Also, $m \text{ of } E_1(\mathit{strid}) \subseteq M_{i-1}$ (by definition of M_i), so $\mathrm{skel}(E(\mathit{strid})) \sqsubseteq \mathrm{Below}(A_2, \varphi^* M_{i-1}) \sqsubseteq (A_2', \varphi^* A_1)$, by the induction hypothesis. Thus

$$(m, \mathrm{skel}(SE \text{ of } E)) \sqsubseteq (A_2', \varphi^* A_1) \tag{6.17}$$

Next, let *id* be an type constructor or a functor identifier in $\mathrm{Dom}\, E$. Since A_1 covers A_2 on N there exists an environment E_1 such that (m, E_1) occurs free in A_1 and $\mathit{id} \in \mathrm{Dom}\, E_1$ and $(m, \varphi^* E_1) = \varphi^*(m, E_1) \sqsubseteq \varphi^* A_1$. By this and 6.17 the domain requirements of cover are satisfied. Let (θ, CE) be a type-structure occuring free in $(m, \mathrm{skel}\, E)$. If it occurs in $\mathrm{skel}(m, SE \text{ of } E)$ there exists a type-structure (θ, CE_1) in $(A_2', \varphi^* A_1)$ with $\mathrm{Dom}\, CE = \mathrm{Dom}\, CE_1$ or $\mathrm{Dom}\, CE = \emptyset$ by the induction hypothesis. Otherwise there exists a $\mathit{tycon} \in \mathrm{Dom}\, E$, such that $E(\mathit{tycon}) = (\theta, CE)$ and as A_1 covers A_2 on N there exists an E_1 such that (m, E_1) occurs free in A_1 and $E_1(\mathit{tycon}) = (\theta, CE_1)$ for some CE_1 with $\mathrm{Dom}\, CE_1 = \mathrm{Dom}\, CE$ or $\mathrm{Dom}\, CE = \emptyset$. As $\varphi^*(m, E_1) \sqsubseteq \varphi^* A_1$ and $\varphi^* A_1 \sqsubseteq A_2$ and A_2 is consistent $E(\mathit{tycon}) = (\varphi^* E_1)(\mathit{tycon})$; hence $(\theta, \varphi^* CE_1)$ occurs free in $\varphi^* A_1$ and the domain requirements are satisfied as $\mathrm{Dom}(\varphi^* CE_1) = \mathrm{Dom}(CE_1)$. We have now proved that $(m, \mathrm{skel}\, E) \sqsubseteq (A_2', \varphi^* A_1)$ in this case.

Case 2, $m_i \in \mathrm{strnames}\, A'_1$. Since $m \in \mathrm{below}(A_2, \varphi^* m_i)$ we then have $(m, \mathrm{skel}\, E) \sqsubseteq A'_2$, by the definition of A'_2. This proves $A_2 \sim (A'_2, \varphi^* A_1)$.

That $N \cap \mathrm{names}\, A'_2 = \emptyset$ is seen as follows. Assume $N \cap \mathrm{names}\, A'_2 \neq \emptyset$, and let m be a structure name which is an element of $N \cap \mathrm{names}\, A'_2$. Since $m \in \mathrm{names}\, A'_2$, there exists an $m' \in \mathrm{names}\, A'_1$ such that $m \in \mathrm{below}(A_2, \varphi^* m')$. Then, since $O_2 \xrightarrow{\psi} O_3$ we have $\psi m \in \mathrm{below}(A_3, \psi\varphi^* m')$, i.e. $m \in \mathrm{below}(A_3, \varphi m')$, since (6.15) commutes and all the realisations are fixed on N. Since $m' \notin N$ and $\varphi A'_1 \sqsubseteq A'_3$ we have $\varphi m' \in \mathrm{names}\, A'_3$. Since $A'_3 \trianglelefteq A_3$ we therefore have $\mathrm{below}(A_3, \varphi m') \subseteq \mathrm{names}\, A'_3$, so $m \in \mathrm{names}\, A'_3$. But $m \in \mathrm{names}(A'_3) \cap N$ contradicts the assumption $N = \mathrm{names}\, A_3 \setminus \mathrm{names}\, A'_3$. Now let t be a type name which is an element of $N \cap \mathrm{names}\, A'_2$. First assume $t \in \mathrm{names}(\varphi^* A'_1)$. Then there exists a t_1 such that $t_1 \in \mathrm{names}\, A'_1$ and $t = \varphi^* t_1$, but as $\varphi A'_1 \sqsubseteq A'_3$ we have $\varphi t_1 \in \mathrm{names}\, A'_3$. By commativity of (6.15) we have $\psi(\varphi^* t_1) = \varphi t_1$, thus $\psi t = \varphi t_1$ and since ψ is fixed on N we then have $t = \varphi t_1$. Thus $t \in \mathrm{names}\, A'_3 \cap N$ which contradicts the assumption $N = \mathrm{names}\, A_3 \setminus \mathrm{names}\, A'_3$. Secondly assume that there exists a $m' \in \mathrm{names}\, A'_1$ such that $t \in \mathrm{names}(\mathrm{Below}(A_2, \varphi^* m'))$. Then since $O_2 \xrightarrow{\psi} O_3$ we have $\psi t \in \mathrm{names}(\mathrm{Below}(A_3, \psi\varphi^* m'))$, i.e. $t \in \mathrm{names}(\mathrm{Below}(A_3, \varphi m'))$ as (6.15) commutes and all the realisations are fixed on N. Since $m' \notin N$ and $\varphi A'_1 \sqsubseteq A'_3$ we have $\varphi m' \in \mathrm{names}\, A'_3$. Since $A'_3 \trianglelefteq A_3$ we therefore have $\mathrm{names}(\mathrm{Below}(A_3, \varphi m')) \subseteq \mathrm{names}\, A'_3$, so $t \in \mathrm{names}\, A'_3$. But $t \in \mathrm{names}\, A'_3 \cap N$ contradicts the assumption $N = \mathrm{names}\, A_3 \setminus \mathrm{names}\, A'_3$. □

Lemma 6.2.11 (Groundedness) *Assume $O \xrightarrow{\varphi} O'$ and that O' is grounded. Then φO is an object of K and grounded.*

Proof Let $(A, B) = O$ and $(A', B') = O'$. Then as $A \sqsupseteq A$ and φA is admissible (by definition of object and morphism in K), we have by Lemma 5.2.3 that $\varphi A \sqsupseteq \varphi B$, so φO is an object of K as φA is type-explicit since A is type-explicit. As $B \,\mathrm{of}\, \varphi O = \varphi B = B'$ we have that $B \,\mathrm{of}\, \varphi O$ is grounded. Now let (θ, CE) be an arbitrary type-structure occuring in A of $\varphi O = \varphi A$ but not within a functor signature. As an assembly consists of structures and type-structures only, (θ, CE) occurs free. Hence by definition of cover, (θ, CE') occurs free in A' for some CE'. Thus as A' is grounded in N of $B' = N$ of φB we have that either θ is simply a type name or $\mathrm{tynames}(\theta) \subseteq N$ of φB; hence φA is grounded in N of φB as required. □

The following theorem states the existence of principal elaborations. It is the main theorem of this paper; item 2 in the theorem corresponds to a theorem in [16].

Theorem 6.2.1 (Principal Elaborations)

1. *Let phrase be one of ty, valdesc or exdesc. For all* P*,* Q'*,* $O = (A, B)$ *and* $O' = (A', B')$*, if* $O \xrightarrow{\varphi} O'$ *and* O *is grounded and*

$$A \sqsupseteq P$$
$$\varphi P \vdash phrase \Rightarrow Q'$$
$$\varphi A \sqsupseteq Q'$$

then there exist $O^* = (A^*, B^*)$*,* φ^* *and* Q^* *depending only on* O*,* P *and phrase, such that*

$$O \xrightarrow{\varphi^*} O^*$$
$$O^* \text{ is grounded}$$
$$\varphi^* A \sqsupseteq (\varphi^* P, Q^*)$$
$$\varphi^* P \vdash phrase \Rightarrow Q^*$$

Moreover, there exists a ψ *such that the diagram*

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

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

2. *Let phrase be one of sigexp, spec, exdesc, strdesc, fundesc, shareq, funsigexp. For all* $O = (A, B)$*,* $O' = (A', B')$*,* Q' *and* φ*, if* $O \xrightarrow{\varphi} O'$ *and* O *is grounded and* $O' \vdash phrase \Rightarrow Q'$ *from* $\mathcal{MR}$ *then there exist* O^**,* φ^* *and* Q^* *depending only on* O *and phrase, such that*

$$O \xrightarrow{\varphi^*} O^*$$
$$O^* \vdash phrase \Rightarrow Q^*$$
$$O^* \text{ is grounded}$$

Moreover, there exists a ψ *such that the diagram*

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

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

Notice that the theorem does not consider the phrases *typdesc*, *datdesc* and *condesc*. The reason is that we do not need these cases to do the induction proof, and we are not particularly interested in what the theorem should say about these phrases — we are mainly interested in the existence of principal elaborations for signature expressions. Moreover, as we shall see below, the proof cases for (eq)type specifications and datatype specifications, shows how an algorithm inferring principal signatures must operate for these phrases.

Actually there is a problem with this formulation. In the case for functor signature expressions we have formally speaking made a logical flaw, see the footnote on page 78. This is straightforward to repair, though time consuming, by formulating the theorem as a completeness property of algorithm W (which we present in the next chapter) as is done in [16] (our formulation follows [15] which contains the same logical flaw). For time considerations I have chosen to defer correcting this problem to future work — by comparing those of our proof cases which have corresponding cases in [16] one can easily be convinced that it is not difficult to correct (it is due to the amount of detail that it is time consuming).

To ease the proof of the first part of the theorem, we shall use the concept of a *structural constraction*, which is defined almost as in the Commentary.

Definition 6.2.3 (Structural mapping) *Let $\mathcal{E}$ be a map over semantic objects, with domain* $\mathrm{Dom}\,\mathcal{E}$*. $\mathcal{E}$ is structural if, for all realisations φ and semantic objects $P \in \mathrm{Dom}\,\mathcal{E}$,*

$$\varphi(\mathcal{E}(P)) = \mathcal{E}(\varphi(P))$$

Definition 6.2.4 (Contraction) *Let $\mathcal{E}$ be a map over semantic objects, with domain* $\mathrm{Dom}\,\mathcal{E}$*. $\mathcal{E}$ is a contraction if, for all $P \in \mathrm{Dom}\,\mathcal{E}$, if P is admissible and grounded in N then*

1. *$(P, \mathcal{E}(P))$ is admissible and grounded in N*

2. *For every structure (m, E') occurring free in $\mathcal{E}(P)$ there exists an E such that (m, E) occurs free in P and*

$$\begin{array}{l}
\mathrm{Dom}(F \text{ of } E') \supseteq \mathrm{Dom}(F \text{ of } E) \\
\mathrm{Dom}(SE \text{ of } E') \supseteq \mathrm{Dom}(SE \text{ of } E) \\
\mathrm{Dom}(TE \text{ of } E') \supseteq \mathrm{Dom}(TE \text{ of } E) \\
\mathrm{Dom}(VE \text{ of } E') \supseteq \mathrm{Dom}(VE \text{ of } E) \\
\mathrm{Dom}(EE \text{ of } E') \supseteq \mathrm{Dom}(EE \text{ of } E)
\end{array}$$

3. *For every type structure (θ, CE') which occurs free in $\mathcal{E}(P)$ there exists a CE such that (θ, CE) occurs free in P and either $CE' = \{\}$ or* $\mathrm{Dom}\, CE' = \mathrm{Dom}\, CE$.

Lemma 6.2.12 *Let $\mathcal{E}$ be a contraction and A an assembly. Then for all $P \in \mathrm{Dom}\, \mathcal{E}$, if $A \sqsupseteq P$ then $A \sqsupseteq (\mathcal{E}(P), P)$.*

Proof Immediate from the definition of admissible cover. □

The point of the above definitions is that many inference rules are (ignoring side-conditions) of the form

$$\frac{\mathcal{E}_1(P) \vdash phrase_1 \Rightarrow Q_1 \quad \cdots \quad \mathcal{E}_k(P, Q_1, \ldots, Q_{k-1}) \vdash phrase_k \Rightarrow Q_k}{P \vdash phrase \Rightarrow \mathcal{E}(P, Q_1, \ldots, Q_k)}$$

for $k \geq 0$, where $\mathcal{E}_1, \ldots, \mathcal{E}_k$ and $\mathcal{E}$ are structural contractions and every premise is inferred by one of the rules listed above. For brevity we shall refer to such a rule as a structural contraction, provided also that its side-conditions depend only on phrase and provided that if $A \sqsupseteq P$ and $A \sqsupseteq \mathcal{E}(P, Q_1, \ldots, Q_k)$ then $A \sqsupseteq Q_1, \ldots, A \sqsubseteq Q_k$.

Some rules contain side-conditions purely for notational convenience and are equivalent to structural contractions without side-conditions. This applies to rules 47 and 49 in the Definition and to rules 4.8 and 4.13.

We now assert that the rules 47–52 in the Definition, 4.4, 4.8, 4.9, 4.10, 4.11–4.13, 4.15–4.17, 4.21–4.23, 4.26 are all structure contractions. (Besides the notes in the Commentary, for rule 4.12 notice that if $A' \sqsupseteq (A, B)$ and $A' \sqsupseteq E_2$ then $A' \sqsupseteq E_1$ as $A \sqsupseteq E_1$ by the first premise.)

Proof We first prove part 1 by induction on the depth of the inference tree. The inference tree for $\varphi P \vdash phrase \Rightarrow Q$ concludes with one of the rules 47–52 in the Definition or 4.17 or 4.21; these rules are all structural contractions and members of $\mathcal{CR}$.

Structural Contractions

Without loss of generality it will be sufficient to deal with the structural contraction rules just in the case $k = 2$. Therefore, consider the case in which the inference tree concludes with an instance

$$\frac{\mathcal{E}_1(\varphi P) \vdash phrase_1 \Rightarrow Q_1 \qquad \mathcal{E}_2(\varphi P, Q_1) \vdash phrase_2 \Rightarrow Q_2}{\varphi P \vdash phrase \Rightarrow \mathcal{E}(\varphi P, Q_1, Q_2)} \tag{6.18}$$

where $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}$ are structural contractions and assume $O = (A, B)$ and $O' = (A', B')$ and

$$O \xrightarrow{\varphi} O' \tag{6.19}$$

$$O \text{ is grounded} \tag{6.20}$$

$$A \sqsupseteq P \tag{6.21}$$

$$\varphi A \sqsupseteq \mathcal{E}(\varphi P, Q_1, Q_2) \tag{6.22}$$

Let $P_1 = \mathcal{E}_1(P)$, $\varphi_1 = \varphi$. Since $\mathcal{E}_1$ is a contraction and $A \sqsupseteq P_1$ we get, by Lemma 6.2.12:

$$A \sqsupseteq P_1 \tag{6.23}$$

Since $\varphi_1 A \sqsupseteq Q$ we get, by definition of structural contraction, that

$$\varphi_1 A \sqsupseteq A_1 \tag{6.24}$$

and by (6.19) we have

$$O \xrightarrow{\varphi_1} O' \tag{6.25}$$

so by induction there exist $O_1^* = (A_1^*, B_1^*)$, φ_1^* and Q_1^* depending only on O, P_1 and $phrase_1$, such that

$$O \xrightarrow{\varphi_1^*} O_1^* \tag{6.26}$$

$$\varphi_1^* A \sqsupseteq (\varphi_1^* P_1, Q_1^*) \tag{6.27}$$

$$\varphi_1^* P_1 \vdash phrase_1 \Rightarrow Q_1^* \tag{6.28}$$

$$O_1^* \text{ is grounded} \tag{6.29}$$

Moreover, for some φ_2, the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\varphi_2"] & & O' \end{tikzcd} \tag{6.30}$$

commutes and

$$\varphi_2 Q_1^* = Q_1 \tag{6.31}$$

We now wish to apply induction to the second premise. Let $P_2 = \mathcal{E}_2(\varphi_1^* P, Q_1^*)$ and let $A_2 = \varphi_1^* A$, $B_2 = \varphi_1^* B$. Then $O_2 = (A_2, B_2) \in \mathrm{Obj}$ and O_2 is grounded by Lemma 6.2.11 and $O \xrightarrow{\varphi_2} O'$ and the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O_2 \arrow[rr, "\varphi_2"] & & O' \end{tikzcd} \tag{6.32}$$

commutes, as (6.30) commutes. As $A_2 \sqsupseteq (\varphi_1^* P_1, Q_1^*)$ by (6.27) and since $\mathcal{E}$ is a contraction

$$A_2 \sqsupseteq P_2 \tag{6.33}$$

Moreover, as $\varphi A \sqsupseteq Q$ by (6.22) and (6.32) commutes we have that $\varphi_2 A_2 \sqsupseteq Q$. Hence by definition of structural contraction

$$\varphi_2 A_2 \sqsupseteq Q_2 \tag{6.34}$$

Also,

$$\begin{array}{rcll} \mathcal{E}_2(\varphi P, Q_1) & = & \mathcal{E}_2(\varphi_2 \varphi_1^* P, \varphi_2 Q_1^*) & \text{by (6.21) and commutativity of (6.32) and by (6.31)} \\ & = & \varphi_2(\mathcal{E}_2(\varphi_1^* P, Q_1^*) & \text{as } \mathcal{E}_2 \text{ is structural} \\ & = & \varphi_2 P_2 & \end{array}$$

so by (6.18) we have

$$\varphi_2 P_2 \vdash \mathit{phrase}_2 \Rightarrow Q_2 \tag{6.35}$$

Thus by induction there exist $O_2^* = (A_2^*, B_2^*)$, φ_2^* and Q_2^* depending only on O_2^*, P_2 and phrase_2, such that

$$O_2 \xrightarrow{\varphi_2^*} O_2^* \tag{6.36}$$

$$\varphi_2^* A_2 \sqsupseteq (\varphi_2^* P_2, Q_2^*) \tag{6.37}$$

$$\varphi_2^* P_2 \vdash \mathit{phrase}_2 \Rightarrow Q_2^* \tag{6.38}$$

$$O_2^* \text{ is grounded} \tag{6.39}$$

Moreover, for some ψ the diagram

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\varphi_2"] & \\ O_2^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.40}$$

commutes and

$$\psi Q_2^* = Q_2 \tag{6.41}$$

Collecting the results: We shall now prove that if we define $\varphi^* = \varphi_2^* \circ \varphi_2^*$, $O^* = O_2^*$ and $Q^* = \mathcal{E}(\varphi^* P, \varphi_2^* Q_1, Q_2^*)$ then

$$O \xrightarrow{\varphi^*} O^*$$

$$\varphi^* A \sqsupseteq (\varphi^* P, Q^*)$$

$$\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*$$

$$O^* \text{ is grounded}$$

and that there exists a ψ with the desired property. First $O \xrightarrow{\varphi^*} O^*$ follows from (6.32) and (6.40), and O^* is grounded by (6.39) and the definition of O^*. We have

$$\varphi^* A \sqsupseteq (\varphi^* P, \varphi_2^* Q_1^*) \tag{6.42}$$

by Lemma 5.2.3 on (6.27) and φ_2^*. By (6.37) we have $\varphi^* A \sqsupseteq Q_2^*$. Thus $\varphi^* A \sqsupseteq (\varphi^* P, \varphi_2^* Q_1, Q_2^*)$ and by Lemma 6.2.12 we get

$$\varphi^* A \sqsupseteq \mathcal{E}(\varphi^* P, \varphi_2^* Q_1, Q_2^*) = Q^* \tag{6.43}$$

and by (6.42) and (6.43) we get the desired

$$\varphi^* A \sqsupseteq (\varphi^* P, Q^*) \tag{6.44}$$

Now we have to check that $\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*$ holds. We first want to show that $\varphi^* P \triangleright \mathit{phrase} \Rightarrow Q^*$ by an instance of the same rule that yielded (6.18). By (6.27) and as $\varphi_2^*(\varphi_1^* P, Q_1^*)$ is admissible by (6.42) we have by the Realisation Theorem that

$$\varphi^* P_1 \vdash \mathit{phrase}_1 \Rightarrow \varphi_2^* Q_1^*$$

Since $\mathcal{E}_1$ is structural and $P_1 = \mathcal{E}_1(P)$ this is equivalent to

$$\mathcal{E}_1(\varphi^* P) \vdash \mathit{phrase}_1 \Rightarrow \varphi_2^* Q_1^* \tag{6.45}$$

Similarly, because $\mathcal{E}_2$ is structural and $P_2 = \mathcal{E}_2(\varphi_1^* P, Q_1^*)$, (6.38) is equivalent to

$$\mathcal{E}_2(\varphi^* P, \varphi_2^* Q_1^*) \vdash \mathit{phrase}_2 \Rightarrow Q_2^* \tag{6.46}$$

From (6.45) and (6.46) we have that

$$\varphi^* P \triangleright \mathit{phrase} \Rightarrow Q^*$$

by an instance of the structural contraction rule under consideration. Moreover, $(\varphi^* P, Q^*)$ is admissible by (6.44) and the side-conditions on the structural contraction rule under consideration, if present, are satisfied because they depend on phrase only and were satisfied at (6.18), so we have the desired

$$\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*$$

It remains to exhibit a ψ with the desired properties. Take any ψ satisfying (6.40) and (6.41). By commutativity of (6.32) and (6.40) we have that

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\psi"] & \\ O_2^* \arrow[rr, "\varphi"] & & O' \end{tikzcd} \tag{6.47}$$

commutes, as required. Moreover,

$$\begin{aligned} \psi Q^* &= \psi(\mathcal{E}(\varphi^* P, \varphi_2^* Q_1^*, Q_2^*)) \\ &= \mathcal{E}(\psi\varphi^* P, \psi\varphi_2^* Q_1^*, \psi Q_2^*) \\ &= \mathcal{E}(\varphi P, Q_1, Q_2) && \text{as (6.40) and (6.47) commutes and by (6.31) and (6.41)} \\ &= Q \end{aligned}$$

as required.

We have now proved part 1 one the theorem. Part 2 is also proved by induction on the depth of the inference. We have not used the used the concept of structural contraction to prove some of the individual cases together — actually there is no good reason for this; we think that it is possible, and that it could shorten the proof. The proof-cases corresponding to rules 4.1, 4.2, 4.3 are almost as in [16] and the proof-case for rule 4.27 and 4.11 are extended versions of the corresponding proof-cases in [16].

$\boxed{\text{Rule 4.1, } \mathit{sigexp} \equiv \texttt{sig}\ \mathit{spec}\ \texttt{end}}$

Assume $O \xrightarrow{\varphi} O'$, O is grounded and $O' \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m', E')$. Then $O' \vdash \mathit{spec} \Rightarrow E'$. By induction there exist φ_1^*, E_1^* and $O_1^* = (A_1^*, B_1^*)$ depending on O and spec only such that $O \xrightarrow{\varphi_1^*} O_1^*$ and $O_1^* \vdash \mathit{spec} \Rightarrow E_1^*$ and O_1^* is grounded. Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.48}$$

commutes and $\psi_1 E_1^* = E'$. Let m^* be a fresh structure name. Let $(A_1^*, B_1^*) = O_1^*$, $A^* = ((m^*, E_1^*), A_1^*)$, $O^* = (A^*, B_1^*)$, $\varphi^* = \varphi_1^*$ and $S^* = (m^*, E_1^*)$. Note that O^* is admissible because m^* is fresh. By $O \xrightarrow{\varphi_1^*} O_1^*$ and the definition of A^* we have $O \xrightarrow{\varphi^*} O^*$ as required. From $O_1^* \vdash \mathit{spec} \Rightarrow E_1^*$ and $O_1^* \sqsubseteq O^*$ we get $O^* \vdash \mathit{spec} \Rightarrow E_1^*$, by Theorem 5.2.1. Thus $O^* \triangleright \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow S^*$, by rule 4.1, and since $O^* \sqsupseteq S^*$ we then have $O^* \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow S^*$, as desired. Moreover, O^* is grounded as O_1^* is grounded and E_1^* is grounded (since $A_1^* \sqsupseteq E_1^*$). Finally, let $\psi = \psi_1 + \{m^* \mapsto m'\}$. Since (6.48) commutes and m^* is fresh, the diagram

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

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

$\boxed{\text{Rule 4.2, } \mathit{sigexp} \equiv \mathit{sigid}}$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{sigid} \Rightarrow S'$ and O is grounded. Let $(A, B) = O$ and $(A', B') = O'$. Then $\mathit{sigid} \in \mathrm{Dom}\, B'$ and $B'(\mathit{sigid}) \geq S'$. Then $\mathit{sigid} \in \mathrm{Dom}\, B$. Write $B(\mathit{sigid})$ in the form $(N^*)S^*$, where $N^* \cap \mathrm{names}\, O = \emptyset$. Let $\varphi^* = \mathrm{Id}$ and let $O^* = (A^*, B)$, where $A^* = (S^*, A)$. Clearly, O^* is admissible and grounded and $O \xrightarrow{\varphi^*} O^*$, as required. Moreover, $(N^*)S^* \geq S^*$ and $A^* \sqsupseteq S^*$, so $O^* \vdash \mathit{sigid} \Rightarrow S^*$. Since $B'(\mathit{sigid}) \geq S'$ we have $\varphi((N^*)S^*) \geq S'$. Thus there exists a ψ such that $\psi n = \varphi n$, for all $n \in \mathrm{names}\, O$ and $\psi S^* = S'$. Thus the diagram

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

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

Rule 4.3, principal signatures

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{sigexp} \Rightarrow \Sigma'$ and O is grounded. Let $(A', B') = O'$. Since $O' \vdash \mathit{sigexp} \Rightarrow \Sigma'$ must be inferred by rule 4.3, there exist N', S' and A'_1 such that $\Sigma' = (N')S'$, $N' \cap \mathrm{names}\, A' = \emptyset$, $A' \trianglelefteq A'_1$, $A'_1, B' \vdash \mathit{sigexp} \Rightarrow S'$, and $(N')S'$ is principal for sigexp in O'. Note that $N' \setminus \mathrm{names}\, S' = \emptyset$, as Σ' is well-formed. We now wish to use the induction hypothesis to prove that sigexp elaborates to a principal signature Σ^* in some O^*. 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 $A'_1, B' \vdash \mathit{sigexp} \Rightarrow S'$, $O \xrightarrow{\varphi} O'_1$ and groundedness of O there exist O^*_1, φ^* and S^* depending only on O and sigexp such that $O \xrightarrow{\varphi^*} O^*_1$ and $O^*_1 \vdash \mathit{sigexp} \Rightarrow S^*$ and O^*_1 is grounded. Moreover, there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^*_1 \arrow[rr, "\psi"] & & O'_1 \end{tikzcd} \tag{6.49}$$

commutes and $\psi S^* = S'$. Let $(A^*_1, B^*_1) = O^*_1$, $A^* = \mathrm{Below}(A^*_1, \varphi^* A)$, $O^* = (A^*, B^*_1)$ and let N^* and Σ^* be given by $\Sigma^* = (N^*)S^* = \mathrm{Clos}_{A^*} S^*$. By Lemma 6.2.9 we have $A^* \trianglelefteq A^*_1$, $A^*_1, B^*_1 \vdash \mathit{sigexp} \Rightarrow S^*$ and Σ^* is principal for sigexp in O^*. Since $O \xrightarrow{\varphi^*} O^*_1$ and $A^* = \mathrm{Below}(A^*_1, \varphi^* A)$ we have the desired $O \xrightarrow{\varphi^*} O^*$, by Lemma 6.1.3. Also, O^* is grounded as O^*_1 is grounded. Next we want to prove that $O^* \vdash \mathit{sigexp} \Rightarrow \Sigma^*$. This almost follows from the results of applying Lemma 6.2.9 above, but we have to prove that $A^* \sqsupseteq \Sigma^*$. Now Σ^* is well-formed (by Lemma 6.2.5 on $A^* \trianglelefteq A^*_1$ and $A^*_1 \sqsupseteq S^*$). But then $A^* \sqsupseteq \Sigma^*$ follows from the definition of Σ^* and the fact that $A^* \trianglelefteq A^*_1$ and $A^*_1 \sqsupseteq S^*$. Thus $O^* \vdash \mathit{sigexp} \Rightarrow \Sigma^*$ holds, as required.

Finally, let ψ be the realisation given at (6.49). We want to prove that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.50}$$

commutes and $\psi \Sigma^* = \Sigma'$. But (6.50) commutes because (6.49) commutes. By Lemma 6.2.6 we have $\Sigma' = \mathrm{Clos}_{A'} S'$. By Lemma 6.2.7 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 4.4, $spec \equiv \texttt{val}\ valdesc$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash spec \Rightarrow \mathrm{Clos}VE$ in Env and that O is grounded. Let $(A, B) = O$ and $(A', B') = O'$. Then $B' = \varphi B$ and $C \mathrm{of} B' \vdash valdesc \Rightarrow VE$. We have that $A \sqsupseteq C \mathrm{of} B$. Since φA is admissible and as there are no type-structures or structures in a VE, φA vacously covers VE, so $\varphi A \sqsupseteq VE$. Thus by part 1 of the theorem there exist $O_1^* = (A_1^*, B_1^*)$, φ_1^* and VE^* depending only on O and *valdesc* such that

$$O \xrightarrow{\varphi_1^*} O_1^* \tag{6.51}$$

$$\varphi_1^* A \sqsupseteq (\varphi_1^*(C \text{ of } B), VE^*) \tag{6.52}$$

$$\varphi_1^*(C \text{ of } B) \vdash valdesc \Rightarrow VE^* \tag{6.53}$$

$$O_1^* \text{ is grounded} \tag{6.54}$$

Also, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.55}$$

commutes and $\psi_1 VE^* = VE$. Let $O^* = O_1^*$, $\varphi^* = \varphi_1^*$ and $E^* = \mathrm{Clos}VE^*$ in Env; these so defined objects clearly all depend only on O and *spec*. Then $O \xrightarrow{\varphi^*} O^*$ and O^* is grounded and $O^* \triangleright spec \Rightarrow E^*$ by (6.53) and rule 4.4 and as $\varphi^*(C \mathrm{of} B) = C \mathrm{of} B^*$. Moreover, $A^* \sqsupseteq E^*$ as $A^* \sqsupseteq VE^*$ by (6.52) and $A^* \sqsupseteq \varphi^* A$. Hence $O^* \vdash spec \Rightarrow E^*$, as required. The required diagram commutativity holds by (6.55). Finally, let $\psi = \psi_1$. Then

$$\begin{aligned} \psi(E^*) &= \psi(\mathrm{Clos}VE^* \text{ in Env}) \\ &= \psi(\mathrm{Clos}VE^*) \text{ in Env} \\ &= \mathrm{Clos}(\psi VE^*) \text{ in Env} \\ &= \mathrm{Clos}VE \text{ in Env} \end{aligned}$$

as required.

Rule 4.5, $spec \equiv \texttt{type}\ typdesc$

Rule 4.5 is easy dealt with by adapting the treatment of rule 4.6 below, simply by omitting (6.60) and replacing (6.64) by

$$t_i^* \text{ does not admit equality, for all } i = 1, \ldots, n$$

$\boxed{\text{Rule 4.6, } \mathit{spec} \equiv \texttt{eqtype } \mathit{typdesc}}$

Let $(A, B) = O$ and $(A', B') = O'$ and assume

$$O \xrightarrow{\varphi} O' \tag{6.56}$$

$$O' \vdash \mathit{spec} \Rightarrow \mathit{TE} \text{ in Env} \tag{6.57}$$

$$O \text{ is grounded} \tag{6.58}$$

Then

$$C \text{ of } B' \vdash \mathit{typdesc} \Rightarrow \mathit{TE} \tag{6.59}$$

$$\forall(\theta, \mathit{CE}) \in \mathrm{Ran}\, \mathit{TE},\ \ \theta \text{ admits equality} \tag{6.60}$$

Write *typdesc* in the form

$$\alpha^{k_1}\mathit{tycon}_1 \texttt{ and } \cdots \texttt{ and } \alpha^{k_n}\mathit{tycon}_n, \quad (n \geq 1)$$

where $\mathit{tycon}_1, \ldots, \mathit{tycon}_n$ are distinct by the syntactic restrictions in Section 2.1. By rule 4.18, there exist $\theta_1, \ldots, \theta_n$ such that

$$\mathit{TE} = \{\mathit{tycon}_1 \mapsto (\theta_1, \{\}), \ldots \mathit{tycon}_n \mapsto (\theta_n, \{\})\} \tag{6.61}$$

$$\mathrm{arity}(\theta_i) = k_i, \text{ for all } i = 1, \ldots, n \tag{6.62}$$

Let $t_1^*, \ldots, t_n^*$ be distinct type names such that

$$\mathrm{arity}(t_i^*) = k_i, \text{ for all } i = 1, \ldots, n \tag{6.63}$$

$$t_i^* \text{ admits equality, for all } i = 1, \ldots, n \tag{6.64}$$

$$\{t_1^*, \ldots, t_n^*\} \cap \mathrm{names}\, A = \emptyset \tag{6.65}$$

Let $\varphi^* = \mathrm{Id}$, $\mathit{TE}^* = \{\mathit{tycon}_1 \mapsto (t_1^*, \{\}), \ldots, \mathit{tycon}_n \mapsto (t_n^*, \{\})\}$ and $E^* = \mathit{TE}^*$ in Env. Also, let $A^* = A \cup \mathit{TE}^*$ and $B^* = B$. Then $O \xrightarrow{\varphi^*} O^*$. Moreover, O^* is grounded by (6.58) and as, for all (θ, CE), θ is a type name in TE^*. By 4.18

$$C \text{ of } B^* \vdash \mathit{typdesc} \Rightarrow \mathit{TE}^*$$

and by rule 4.6 and as $A^* \sqsupseteq \mathit{TE}^*$ (by definition of admissible cover) we have

$$O^* \vdash \mathit{spec} \Rightarrow E^*$$

Clearly, O^*, φ^* and E^* depends only on O and *spec*. Finally, let $\psi = \varphi + \{t_1^* \mapsto \theta_1, \ldots, t_n^* \mapsto \theta_n\}$ which really is a realisation because of (6.62), (6.63) and (6.64). Then

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

commutes, and

$$\begin{aligned}
\psi(E^*) &= \psi TE^* \text{ in Env} \\
&= TE \text{ in Env}
\end{aligned}$$

as required.

$\boxed{\text{Rule 4.7, } spec \equiv \texttt{datatype}\ datdesc}$

Without loss of generality, assume that

$$\begin{aligned}
datdesc &\equiv \alpha^{(k)}tycon \texttt{ = } condesc \\
condesc &\equiv con \text{ of } ty
\end{aligned}$$

(the general case of several type constructors and several value constructors presents no extra difficulty, and neither does the case where "of ty" is absent.) Let $(A, B) = O$ and $(A', B') = O'$ and assume that

$$O \xrightarrow{\varphi} O' \tag{6.66}$$

$$O' \vdash spec \Rightarrow E \tag{6.67}$$

$$O \text{ is grounded} \tag{6.68}$$

Then the inference tree that proves (6.67) is of the form

$$\begin{array}{c}
\nabla_1 \\
\Big|\ (4.20) \\
C \text{ of } B' + TE, \alpha^{(k)}t \vdash \boxed{con \text{ of } ty} \Rightarrow \underbrace{\{con \mapsto \tau \to alpha^{(k)}t\}}_{CE} \\
\Big|\ (4.19) \\
C \text{ of } B' + TE \vdash \boxed{\alpha^{(k)}tycon \texttt{ = } condesc} \Rightarrow VE, TE \\
\Big|\ (4.7) \\
A', B' \triangleright \boxed{\texttt{datatype}\ \alpha^{(k)}tycon \texttt{ = } condesc} \Rightarrow (VE, TE) \text{ in Env}
\end{array}$$

where $VE = \mathrm{Clos}CE$, $TE = \{tycon \mapsto (t, \mathrm{Clos}CE)\}$ and ∇_1 is the inference tree that proves

$$C \text{ of } B' \vdash ty \Rightarrow \tau \tag{6.69}$$

Now choose $t^* \notin \text{names}\, A$, with arity k, not admitting equality. Let $TE_0^* = \{tycon \mapsto (t^*, \{\})\}$. (Note the empty constructor environment, we shall shortly obtain TE^* by "filling out" the constructor environment of TE_0^*.) Let $\psi = \varphi + \{t^* \mapsto t\}$.

Let $TE_0 = \{tycon \mapsto (t, \{\})\}$. Note that C of $B' + TE_0$ is admissible (as C of $B' + TE$ is admissible, c.f. the inference tree above.) Thus we can apply Lemma A.4 of the Commentary [11, Page 119] to (6.69). Hence there exists a ∇_0 which proves C of $B' + TE_0 \vdash ty \Rightarrow \tau$. Note that C of $B + TE_0^*$ is admissible (for consistency, notice that $\mathrm{Dom}(CE \text{ of } TE_0^*) = \{\}$) and that $\psi(C \text{ of } B + TE_0^*) = C \text{ of } B' + TE_0$. Then, by Lemma A.3 (2) of the Commentary [11, Page 119], there exist ∇_0^* and τ^* such that

$$\psi\nabla_0^* = \nabla_0 \tag{6.70}$$

$$\nabla_0^* \text{ proves } C \text{ of } B + TE_0^* \vdash ty \Rightarrow \tau^* \tag{6.71}$$

Now let $TE^* = \{tycon \mapsto (\tau^*, \mathrm{Clos}CE^*)\}$ where $CE^* = \{con \mapsto \tau^* \to \alpha^{(k)}t^*\}$. Note that C of $B + TE^*$ is admissible as $t^* \notin \text{names}\, A$ (and so $t^* \notin \text{names}\, B$). Thus we can apply Lemma A.4 (second half) of the Commentary to (6.71), so there exists a ∇_1^* such that

$$\nabla_1^* \text{ proves } C \text{ of } B + TE^* \vdash ty \Rightarrow \tau^* \tag{6.72}$$

Using that $\psi\tau^* = \tau$, by (6.70), we get

$$\psi(C \text{ of } B + TE^*) = C \text{ of } B + TE^* \tag{6.73}$$

which is admissible. Thus we can apply Lemma A.3 (1) of the Commentary to (6.72); hence

$$\psi\nabla_1^* \text{ proves } C \text{ of } B' + TE \vdash ty \Rightarrow \tau \tag{6.74}$$

By Lemma A.4 (first half) of the Commentary on (6.74) and (6.69) we therefore have

$$\psi\nabla_1^* = \nabla_1 \tag{6.75}$$

Now let $\varphi^* = \mathrm{Id}$ and $\nabla^* =$

$$\begin{array}{c}
\nabla_1^* \\
\Big| \ (4.20) \\
C \text{ of } B + TE^*, \alpha^{(k)}t^* \rhd \boxed{con \text{ of } ty} \Rightarrow \underbrace{\{con \mapsto \tau^* \to \alpha^{(k)}t^*\}}_{CE^*} \\
\Big| \ (4.19) \\
C \text{ of } B + TE^* \rhd \boxed{\alpha^{(k)}tycon \texttt{ = } condesc} \Rightarrow VE^*, TE^* \\
\Big| \ (4.7) \\
A', B' \rhd \boxed{\texttt{datatype } \alpha^{(k)}tycon \texttt{ = } condesc} \Rightarrow E^*
\end{array}$$

where $O^* = (A^*, B^*)$, $A^* = A \cup TE^*$, $VE^* = \mathrm{Clos}VE^*$ and $E^* = (VE^*, TE^*)$ in Env. Notice that $O \xrightarrow{\varphi^*} O^*$ and that O^* is grounded by (6.68) and as TE^* is grounded (because all type functions are type names in TE^*); this together with the fact that C of $B + TE^*$ is admissible and (6.72) means that every $\rhd$ in ∇^* can be turned into a $\vdash$ using rules 4.7, 4.19 and 4.20, such that $O' \vdash spec \Rightarrow E^*$ as required. Moreover, by Lemma A.4 (first half) of the Commentary on (6.72), ∇_1^* depends only on C of $B + TE^*$ and ty. It follows that φ^*, E^* and O^* depend only on O and $spec$ as required. Also,

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

commutes, as required. Finally, since $\psi(t^*, \tau^*) = (t, \tau)$

$$\psi E^* = E$$

as required.

$\boxed{\text{Rule 4.8, } spec \equiv \texttt{exception } exdesc}$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash spec \Rightarrow (VE, EE)$ in Env and that O is grounded. Let $(A, B) = O$ and $(A', B') = O'$. Then C of $B \vdash exdesc \Rightarrow EE$ and $VE = EE$. We have that $A \sqsupseteq C$ of B. Since φA is admissible and as there are no type-structures or structures in an exception environment, $\varphi A \sqsupseteq EE$. Thus by induction there exist $O_1^* = (A^*, B^*)$, φ_1^*, EE^* depending only on O and *exdesc* such that

$$O \xrightarrow{\varphi_1^*} O_1^* \tag{6.76}$$

$$\varphi^* A \sqsupseteq (\varphi_1^*(C \text{ of } B), EE^*) \tag{6.77}$$

$$\varphi^*(C \text{ of } B) \vdash exdesc \Rightarrow EE^* \tag{6.78}$$

$$O_1^* \text{ is grounded} \tag{6.79}$$

Also, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.80}$$

commutes and $\psi_1 EE^* = EE$. Let $O^* = O_1^*$, $\varphi^* = \varphi_1^*$ and $E^* = (EE^*, EE^*)$ in Env; these so defined objects are clearly only dependent on O and *spec*. Then $O \xrightarrow{\varphi^*} O^*$ and O^* is grounded by (6.79), and $O^* \triangleright spec \Rightarrow EE^*$ by (6.78) and rule 4.8. Moreover, since $\varphi^* A \sqsupseteq EE^*$ (by (6.78)) and $A^* \sqsupseteq \varphi^* A$ (by(6.76)) we have $A^* \sqsupseteq EE^*$; hence $O^* \vdash spec \Rightarrow E^*$. The required diagram commutativity holds by (6.80). Finally, let $\psi = \psi_1$. Then

$$\begin{aligned} \psi(E^*) &= \psi(EE^*, EE^*) \text{ in Env} \\ &= (EE, EE) \text{ in Env} \\ &= (EE, VE) \text{ in Env} \end{aligned}$$

as required.

Rule 4.9, $spec \equiv$ `structure` $strdesc$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash spec \Rightarrow E$ and that O is grounded. Let $(A, B) = O$. Then by rule 4.9 there exists a SE such that $E = SE$ in Env and $O' \vdash strdesc \Rightarrow SE$. So by induction there exist O^*, φ^* and SE^* depending only on O and $spec$ such that

$$O \xrightarrow{\varphi^*} O^* \tag{6.81}$$

$$O^* \vdash strdesc \Rightarrow SE^* \tag{6.82}$$

$$O^* \text{ is grounded} \tag{6.83}$$

and, moreover, there exists a ψ such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.84}$$

commutes and $\psi SE^* = SE$.

Let $E^* = SE^*$ in Env. Then by rule 4.9, $O^* \vdash spec \Rightarrow E^*$ and $\psi E^* = E$, which together with (6.81), (6.83) and (6.84) is the required.

Rule 4.10, $spec \equiv$ `functor` $fundesc$

The proof is analogous to the proof for rule 4.9.

Rule 4.11, $spec \equiv$ `sharing` $shareq$

The proof is analogous to the proof for rule 4.9.

$\boxed{\text{Rule 4.12, } spec \equiv \texttt{local } spec_1 \texttt{ in } spec_2 \texttt{ end}}$

Assume $O \xrightarrow{\varphi} O'$ and that O is grounded. Let $(A, B) = O$, $(A', B') = O'$ and assume that $O' \vdash spec \Rightarrow E'_2$. By rule 4.12 there exists an E_1 such that $(A', B') \vdash spec_1 \Rightarrow E'_1$ and $(A', B' + E'_1) \vdash spec_2 \Rightarrow E'_2$. By induction on $O \xrightarrow{\varphi} O'$ and $(A', B') \vdash spec_1 \Rightarrow E'_1$ and groundedness of O, there exists $O^*_1 = (A^*_1, B^*_1)$, E^*_1 and φ^*_1 depending only on O and $spec_1$ such that

$$O \xrightarrow{\varphi^*_1} O^* \tag{6.85}$$

$$O^* \vdash spec_1 \Rightarrow E^*_1 \tag{6.86}$$

$$O^*_1 \text{ is grounded} \tag{6.87}$$

Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*_1"'] \arrow[dr, "\varphi"] & \\ O^*_1 \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.88}$$

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 $O'_2 \vdash spec_2 \Rightarrow E'_2$ and O_2 is grounded (as $A^*_1 \sqsupseteq E^*_1$ and A^*_1 is grounded). Thus by induction there exist $O^*_2 = (A^*_2, B^*_2)$, φ^*_2 and E^*_2 depending only on O_2 and $spec_2$ such that

$$O_2 \xrightarrow{\varphi^*_2} O^*_2 \tag{6.89}$$

$$O^*_2 \vdash spec_2 \Rightarrow E^*_2 \tag{6.90}$$

$$O^*_2 \text{ is grounded} \tag{6.91}$$

Moreover, there exists a ψ_2 such that

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi^*_2"'] \arrow[dr, "\psi_1"] & \\ O^*_2 \arrow[rr, "\psi_2"] & & O'_2 \end{tikzcd} \tag{6.92}$$

commutes and $\psi_2 E^*_2 = E'_2$. Let $O^* = (A^*_2, \varphi^*_2(B^*_1))$, $\varphi^* = \varphi^*_2 \circ \varphi^*_1$. We wish to prove

$$O \xrightarrow{\varphi^*} O^* \tag{6.93}$$

and

$$O^* \vdash spec \Rightarrow E^*_2 \tag{6.94}$$

and

$$O^* \text{ is grounded} \tag{6.95}$$

Since $B_1^* \sqsubseteq A_1^*$ and $\varphi_2^* A_1$ is admissible (as $\varphi_2^* A_1^* \sqsubseteq A_2^*$ by $O_2 \xrightarrow{\varphi_2^*} O_2^*$) we have, by Lemma 5.2.3, that $\varphi_2^* B_1^* \sqsubseteq \varphi_2^* A_1^* \sqsubseteq A_2^*$. Thus O^* is an object in K and $O_1^* \xrightarrow{\varphi_2^*} O^*$ holds. Since (6.92) commutes we then have that

$$\begin{tikzcd} & O_1^* \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O^* \arrow[rr, "\psi_2"] & & O' \end{tikzcd} \tag{6.96}$$

commutes. We get $O \xrightarrow{\varphi^*} O^*$ by composition of (6.88) and (6.96). By (6.86) and (6.90) and rule 4.12 we get (6.94) as required. O^* is grounded as O_2^* is grounded, as desired. Let $\psi = \psi_2$. Since (6.88) and (6.96) commutes, the diagram

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

commutes and, moreover, we have $\psi E_2^* = E_2'$ as required.

Rule 4.13, $spec \equiv \texttt{open}\ longstrid_1 \cdots longstrid_n$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash spec \Rightarrow E'$ and that O is grounded. Let $(A, B) = O$ and $(A', B') = O'$. Then by rule 4.13

$$B'(longstrid_1) = (m_1', E_1') \quad \cdots \quad B'(longstrid_n) = (m_n', E_n')$$

and $E' = E_1' + \cdots + E_n'$. Since $B' = \varphi B$ there exist $(m_1, E_1), \ldots, (m_n, E_n)$ such that

$$B(longstrid_1) = (m_1, E_1) \quad \cdots \quad B(longstrid_n) = (m_n, E_n)$$

Let $E = E_1 + \cdots + E_n$, $O^* = O$, $\varphi^* = \mathrm{Id}$. Then $O \xrightarrow{\varphi^*} O^*$ trivially, O^* is grounded and

$$O^* \vdash spec \Rightarrow E$$

Let $\psi = \varphi$. Then

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

of course trivially commutes, and $\psi E = E'$ as required.

$\boxed{\text{Rule 4.14, } \mathit{spec} \equiv \texttt{include } \mathit{sigid}_1 \cdots \mathit{sigid}_n}$

We will not prove this case directly, instead we simply notice that one can use the observation of Exercise 6.1 in the Commentary — which together with Exercise 3.2 show that the `include` specification could have been given as a derived form (using local and open specifications).

$\boxed{\text{Rule 4.15, } \mathit{spec} \equiv \quad\quad}$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \quad \Rightarrow \{\}$ and that O is grounded. Let $O^* = O$, $\varphi^* = \mathrm{Id}$, $Q^* = \{\}$ and $\psi = \varphi$. Then they obviously satisfy the required.

$\boxed{\text{Rule 4.16, } \mathit{spec} \equiv \mathit{spec}_1 \langle ; \rangle\ \mathit{spec}_2}$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{spec}_1 \ \langle ; \rangle\ \mathit{spec}_2 \Rightarrow E'$ and that O is grounded. Let $(A', B') = O'$. By rule 4.16 there exist E'_1 and E'_2 such that $E' = E'_1 + E'_2$ and $A', B' \vdash \mathit{spec}_1 \Rightarrow E'_1$ and $A', B' + E'_1 \vdash \mathit{spec}_2 \Rightarrow E'_2$. By induction on $O \xrightarrow{\varphi} O'$ and $A', B' \vdash \mathit{spec}_1 \Rightarrow E'_1$ there exists $O^*_1 = (A^*_1, B^*_1)$, E^*_1 and φ^*_1 depending on O and spec_1 only such that $O \xrightarrow{\varphi^*_1} O^*_1$ and $O^*_1 \vdash \mathit{spec}_1 \Rightarrow E^*_1$ and O^*_1 is grounded. Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*_1"'] \arrow[dr, "\varphi"] & \\ O^*_1 \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.97}$$

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 $O'_2 \vdash \mathit{spec}_2 \Rightarrow E'_2$ and O_2 is grounded (as O^*_1 is grounded and E^*_1 is grounded since $A^*_1 \sqsupseteq E^*_1$). Thus by induction there exist $O^*_2 = (A^*_2, B^*_2)$, φ^*_2 and E^*_2, depending on O_2 and spec_2 only, such that $O_2 \xrightarrow{\varphi^*_2} O^*_2$ and O^*_2 is grounded and $O^*_2 \vdash \mathit{spec}_2 \Rightarrow E^*_2$. Moreover, there exits a ψ_2 such that

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi^*_2"'] \arrow[dr, "\psi_1"] & \\ O^*_2 \arrow[rr, "\psi_2"] & & O'_2 \end{tikzcd} \tag{6.98}$$

commutes and $\psi_2 E^*_2 = E'_2$. Let $O^* = (A^*_2, \varphi^*_2(B^*_1))$, $\varphi^* = \varphi^*_2 \circ \varphi^*_1$ and $E^* = \varphi^*_2 E^*_1 + E^*_2$. We wish to prove $O \xrightarrow{\varphi^*} O^*$ and that O^* is grounded and

$$O^* \vdash \mathit{spec}_1 \ \langle ; \rangle\ \mathit{spec}_2 \Rightarrow E^* \tag{6.99}$$

O^* is grounded since O_2^* is grounded and $O_2 \xrightarrow{\varphi_2^*} O_2^*$. Since $B_1^* \sqsubseteq A_1^*$ and $\varphi_2^* A_1^*$ is admissible (as $O_2 \xrightarrow{\varphi_2^*} O_2^*$) we have, by Lemma 5.2.3, $\varphi_2^* B_1^* \sqsubseteq \varphi_2^* A_1^* \sqsubseteq A_2^*$. Thus O^* is an object in K and $O_1^* \xrightarrow{\varphi_2^*} O^*$ holds. Since (6.98) commutes we then have that

$$\begin{tikzcd} & O_1^* \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O^* \arrow[rr, "\psi_2"] & & O' \end{tikzcd} \tag{6.100}$$

commutes. We get $O \xrightarrow{\varphi^*} O^*$ by composition of (6.97) and (6.100). Moreover, by Theorem 5.2.1 on $O_1^* \vdash \mathit{spec}_1 \Rightarrow E_1^*$ and $O_1^* \xrightarrow{\varphi_2^*} O^*$ we have $O^* \vdash \mathit{spec}_1 \Rightarrow \varphi_2^* E_1^*$. Also $O_2^* = (A_2^*, B_2^*) = (A_2^*, \varphi_2^* B_1^* + \varphi_2^* E_1^*)$ (as $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and B of $O_2 = B_1^* + E_1^*$ and the definition of morphishms in K), so from $O_2^* \vdash \mathit{spec}_2 \Rightarrow E_2^*$ we get $O^* + \varphi_2^* E_1^* \vdash \mathit{spec}_2 \Rightarrow E_2^*$. Thus by rule 4.16 we have (6.99) as required. Let $\psi = \psi_2$. Since (6.97) and (6.100) commute, the diagram

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

commutes and moreover, since (6.98) commutes, we have $\psi E^* = \psi(\varphi_2^* E_1^* + E_2^*) = \psi \varphi_2^* E_1^* + \psi E_2^* = \psi_1 E_1^* + \psi E_2^* = E_1' + E_2' = E'$, as required.

Rule 4.22, $\mathit{strdesc} \equiv \mathit{strid}$: sigexp $\langle$ `and` $\mathit{strdesc}$ $\rangle$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{strid}$: sigexp `and` $\mathit{strdesc} \Rightarrow \mathit{SE}$ and that O is grounded. Then by rule 4.22 there exist S' and SE_1' such that $O' \vdash \mathit{sigexp} \Rightarrow S'$ and $O' \Rightarrow \mathit{strdesc} \Rightarrow \mathit{SE}_1'$. Thus by induction there exist O_1^*, φ_1^* and S^* such that $O \xrightarrow{\varphi_1^*} O_1^*$ and $O_1^* \vdash \mathit{sigexp} \Rightarrow S^*$ and O_1^* is grounded. Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.101}$$

commutes, and $\psi_1 S^* = S'$.

Let $O_2 = O_1^*$ and $O_2' = O'$. Then $O_2 \xrightarrow{\psi_1} O_2'$ and $O_2' \vdash \mathit{strdesc} \Rightarrow \mathit{SE}_1'$ and O_2 is grounded. Thus by induction there exist $O_2^* = (A_2^*, B_2^*)$, φ_2^* and SE_1^* such that $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and A_2^* is grounded in N of B and $O_2^* \vdash \mathit{strdesc} \Rightarrow \mathit{SE}_1^*$. Moreover, there exists a ψ_2 such that the diagram

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O_2^* \arrow[rr, "\psi_2"] & & O_2' \end{tikzcd} \tag{6.102}$$

commutes, and $\psi_2 SE_1^* = SE'$.

Let $O^* = O_2^*$, $\varphi^* = \varphi_2^* \circ \varphi_1^*$, $SE^* = \{strid \mapsto \varphi_2^* S^*\} + SE_1^*$. We now want to show $O \xrightarrow{\varphi^*} O^*$ and

$$O^* \vdash strid : sigexp \texttt{ and } strdesc \Rightarrow SE^* \tag{6.103}$$

(We already have that O^* is grounded.) We have $O \xrightarrow{\varphi^*} O^*$ simply by composition of morphisms in K. As $O_1^* \vdash sigexp \Rightarrow S^*$ and $O_1^* \xrightarrow{\varphi_2^*} O_2^*$ we have by Theorem 5.2.1 that $O_2^* \vdash sigexp \Rightarrow \varphi_2^* S^*$. Thus by rule 4.22 we have (6.103). Let $\psi = \psi_2$. Since (6.101) and (6.102) commutes, the diagram

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

commutes and, moreover, since (6.102) commutes, we have

$$\begin{aligned} \psi SE^* &= \{strid \mapsto \psi\varphi_2^* S^*\} + \psi SE^* \\ &= \{strid \mapsto \psi_1 S^*\} + \psi SE^* \\ &= \{strid \mapsto S'\} + SE' \\ &= SE \end{aligned}$$

as required.

Note that we here assumed that "`and` *strdesc*" is present; if this is not the case, the proof is very easy: O^* and φ^* are just O_1^* and φ_1^* obtained by induction on *sigexp*.

$\boxed{\text{Rule 4.23, } \mathit{fundesc} \equiv \mathit{funid}\ \mathit{funsigexp}\ \langle \texttt{and}\ \mathit{fundesc} \rangle}$

This case is analogous to the case for rule 4.22.

Rule 4.24, $spec \equiv$ `sharing` $longstrid_1$ `=` $\cdots$ `=` $longstrid_n$ [1]

Assume $O \xrightarrow{\varphi} O'$. Let $(A, B) = O$ and $(A', B') = O'$ and assume that O is grounded and $O' \vdash$ `sharing` $longstrid_1$ `=` $\cdots$ `=` $longstrid_k \Rightarrow \{\}$ for some $k \geq 2$. By the side-condition on rule 4.24 we have m of $(B'(longstrid_1)) = \cdots = m$ of $(B'(longstrid_k))$, but the problem is that we do not necessarily have m of $(B(longstrid_1)) = \cdots = m$ of $(B(longstrid_k))$. (One reason is that whenever we choose names in the proof, e.g. in the cases for rules 4.1, 4.2 and 4.14, we always choose them to be suitably "new".) Notice also that we cannot just choose $\varphi^* = \varphi$, as φ^* then would not depend only on O and the phrase.

However, let X be any structure identifier, let m and m' be structure names such that $m \notin \text{names}(O)$ and $m' \notin \text{names}(O')$. Consider the assembly

$$A_1 = \Big((m, \{X \mapsto B(longstrid_1)\}), \ldots, ((m, \{X \mapsto B(longstrid_k)\}), A)\Big)$$

Let $N = N$ of B and let $\varphi_1 = \varphi + \{m \mapsto m'\}$. We shall now prove that φ_1 is an admissifier for A_1 under N. By $O \xrightarrow{\varphi} O'$ and the definition of K we have that φ is fixed on N, that $\varphi B = B'$, that $\varphi A \sqsubseteq A'$, that A covers A' on N, and that A' is admissible. Hence φA is admissible and A covers φA on N. Thus φ is an admissifier for A under N. But then, since m of $(B'(longstrid_1)) = \cdots = m$ of $(B'(longstrid_k))$, we have that φ_1 is an admissifier for A_1 under N.

Thus, as A_1 is grounded in N (as O is grounded by assumption), by Theorem 5.1.1 there exists a principal admissifier φ^* for A_1 under N, and $\varphi^* A_1$ is grounded in N, in particular $\varphi^* A$ is grounded in N. We now want to show that $\varphi^* B$ is grounded. First notice that N of $\varphi^* B = N$ as φ^* is fixed on N. Let us assume that $\varphi^* B$ is not grounded. Then there exists a type-structure (θ, CE) occuring in $\varphi^* B$ such that θ is not a type name and $\text{tynames}(\theta) \not\subseteq N$. As φ^* is fixed on N, there must exist some type-structure (θ', CE') occuring in B, such that $\varphi^*(\theta', CE') = (\theta, CE)$ and either (i) θ' is a type name not in N or (ii) θ is not a type name and $\text{tynames}(\theta') \subseteq N$. Assume (ii). Then $\theta = \theta'$ as φ^* is fixed on N so this cannot be the case. Assume (i). It must be the case that θ' is bound by some name set prefix, because otherwise $\theta = \theta'$ by definition of application of a realisation. So assume θ' is not bound by some name set prefix, i.e., that (θ', CE') occurs free in B. Then, for some CE'', the type-structure (θ', CE'') occurs in A, as $A \sqsupseteq B$. Therefore, $\varphi^*(\theta, CE'') = (\theta, \varphi^* CE'')$ occurs in $\varphi^* A$ which is grounded, so either θ is a type name or $\text{tynames}(\theta) \subseteq N$, so (i) cannot be the case either. We therefore conclude that $\varphi^* B$ is grounded.

[1] Almost as in [16]

Let $O^* = (A^*, B^*) = \varphi^*(O)$. We have that O^* is grounded. Since $\varphi^* A_1$ is consistent, we have

$$m \text{ of } (B^*(\mathit{longstrid}_1)) = \cdots = m \text{ of } (B^*(\mathit{longstrid}_k))$$

Thus $O^* \vdash \mathit{spec} \Rightarrow \{\}$, and $\varphi^* A$ is grounded in N as required.

Since φ^* is a principal admissifier for A_1, there exists a ψ such that ψ is fixed on N and $\psi(\varphi^* A_1) = \varphi_1 A_1$. In particular, $\psi(\varphi^* A) = \varphi A$. Thus if the morphisms in

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

exist in K then the diagram commutes. That the left-hand side morphism is in K follows from the fact that φ^* is an admissifier for A_1. That the bottom morphism is in K is seen as follows. We have $\psi(A^*) = \psi(\varphi^* A) = \varphi A \sqsubseteq A'$ (as required by the definition of morphisms in K). Also, $\varphi^* A$ covers A' on N by Lemma 5.1.3 (as required by the definition of morphisms in K).

Rule 4.25, $\mathit{spec} \equiv$ `sharing` $\mathit{longtycon}_1$ `=` $\cdots$ `=` $\mathit{longtycon}_k$

The proof is the same as for rule 4.24, with the following modifications

1. Replace *longstrid* by *longtycon* throughout.
2. Replace "m of" with "θ of" throughout.
3. X is now a type constructor, not a structure identifier.

$\boxed{\text{Rule 4.26, } spec \equiv shareq_1 \texttt{ and } shareq_2}$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash shareq_1 \texttt{ and } shareq_2 \Rightarrow \{\}$ and that O is grounded. Then by rule 4.26 $O' \vdash shareq_1 \Rightarrow \{\}$ and $O' \vdash shareq_2 \Rightarrow \{\}$. Thus by induction there exist O_1^* and φ_1^* depending only on O and $shareq_1$ such that $\emptyset \xrightarrow{\varphi_1^*} O_1^*$ and O_1^* is grounded and $O_1^* \vdash shareq_1 \Rightarrow \{\}$. Moreover there exists a ψ_1 such that

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.104}$$

commutes and $\psi\{\} = \{\}$ (!)

Let $O_2 = O_1^*$ and $O_2' = O'$. Then $O_2 \xrightarrow{\psi_1} O_2'$ and $O_2' \vdash shareq_2 \Rightarrow \{\}$ and O_2 is grounded. Thus by induction there exist O_2^*, φ_2^* such that $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and $O_2^* \vdash shareq_2 \Rightarrow \{\}$ and O_2^* is grounded. Moreover, there exists a ψ_2 such that the following diagram

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O_2^* \arrow[rr, "\psi_2"] & & O_2' \end{tikzcd} \tag{6.105}$$

commutes and $\psi_2\{\} = \{\}$.

Let $O^* = O_2^*$, $\varphi^* = \varphi_2^* \circ \varphi_1^*$. We have that $O \xrightarrow{\varphi^*} O^*$ and O^* is grounded. We now want to show that

$$O^* \vdash shareq_1 \texttt{ and } shareq_2 \Rightarrow \{\} \tag{6.106}$$

By $O_1^* \xrightarrow{\varphi_2^*} O^*$ and $O^* \vdash shareq_1 \Rightarrow \{\}$ we get by Theorem 5.2.1 that $O^* \vdash shareq_1 \Rightarrow \{\}$. Then by rule 4.26 we have (6.106). Let $\psi = \psi_2$. By commutativity of (6.104) and (6.105) the diagram

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

commutes and, moreover, we have $\psi\{\} = \{\}$ as required.

Rule 4.27, *funsigexp* $\equiv$ (*strid* : *sigexp*$_1$) : *sigexp*$_2$

Assume $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{funsigexp} \Rightarrow \Phi'$, where $\mathit{funsigexp} \equiv (\mathit{strid} : \mathit{sigexp}_1) : \mathit{sigexp}_2$ and that O is grounded. Then by rule 4.27, Φ' is of the form $(N_1')(S_1', \Sigma_2')$ and $O' \vdash \mathit{sigexp}_1 \Rightarrow (N_1')S_1'$ and

$$(S_1', A'), B' + N_1' + \{\mathit{strid} \mapsto\ S_1'\} \vdash \mathit{sigexp}_2 \Rightarrow \Sigma_2' \tag{6.107}$$

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

By induction on $O \xrightarrow{\varphi} O'$ and $O' \vdash \mathit{sigexp}_1 \Rightarrow (N_1')S_1'$ and groundedness of O there exist O_1^*, φ_1^* and $\Sigma_1^* = (N_1^*)S_1^*$ depending on O and sigexp_1 only, such that $O \xrightarrow{\varphi_1^*} O_1^*$ and O_1^* is grounded and

$$O_1^* \vdash \mathit{sigexp}_1 \Rightarrow (N_1^*)S_1^* \tag{6.108}$$

Moreover, there exists a ψ_1 such that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi_1^*"'] \arrow[dr, "\varphi"] & \\ O_1^* \arrow[rr, "\psi_1"] & & O' \end{tikzcd} \tag{6.109}$$

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

Without loss of generality, we may assume that $N_1^* \cap \mathrm{names}\, O_1^* = \emptyset$; 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

$$\psi_1((N_1^*)S_1^*) = (\psi_1 N_1^*)\ (\psi_1 S_1^*) = (N_1')S_1' \tag{6.110}$$

Let $(A_1^*, B_1^*) = O_1^*$. Further, let

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

We have $O_2 \xrightarrow{\psi_1} O_2'$. (To see this, first note that O_2' is an object in K by (6.107); similarly, O_2 is an object in K by (6.108). Also, O_2 is grounded, as O^* is grounded and $A^* \sqsupseteq S_1^*$. 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 $B_2 = (N \text{ of } B) \cup N_1^*$, for A_1^* covers A' on N of B so by (6.110) 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 $(N \text{ of } B) \cup N_1^*$.) Also, by (6.107) we have $O_2' \vdash \mathit{sigexp}_2 \Rightarrow \Sigma_2'$. By induction on $O_2 \xrightarrow{\psi_1} O_2'$ and $O_2' \vdash \mathit{sigexp}_2 \Rightarrow \Sigma_2'$ and groundedness of O_2 there exist O_2^*, φ_2^* and Σ_2^* depending only on O_2 and $sigexp_2$ such that $O_2 \xrightarrow{\varphi_2^*} O_2^*$ and O_2^* is grounded and

$$O_2^* \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{6.111}$$

Moreover, there exists a ψ_2 such that the diagram

$$\begin{tikzcd} & O_2 \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O_2^* \arrow[rr, "\psi_2"] & & O_2' \end{tikzcd} \tag{6.112}$$

commutes and $\psi_2 \Sigma_2^* = \Sigma_2'$.

Let $(A_2^*, B_2^*) = O_2^*$. In the proof case concerning rule 4.3, we saw that $A_2^* = \text{Below}(A_0, \varphi_2^* A_2)$, for some A_0[2]. (The construction of $(O_2^*, \varphi_2^*, \Sigma_2^*)$ may widen flexible structures that are already in O_2, but it "introduces" no other structures.) Thus $A_2^* \sqsubseteq \text{Below}(A_2^*, \varphi_2^* A_2)$. Also, we have $\psi_1 A_1^* \sqsubseteq A'$, by (6.109). Let $A^* = \text{Below}(A_2^*, \varphi_2^* A_1^*)$. Note that $A^* \trianglelefteq A_2^*$, by Lemma 6.1.2. Then by Lemma 6.2.10 we have $A_2^* \sim (A^*, \varphi_2^* A_2)$ and

$$N_1^* \cap \text{names}\, A^* = \emptyset \tag{6.113}$$

Since $\varphi_2^* A_2 = (\varphi_2^* S_1^*, \varphi_2^* A_1^*) \sqsubseteq (\varphi_2^* S_1^*, A^*)$ we have $(A^*, \varphi_2^* A_2) \sim (A^*, \varphi_2^* S_1)$, which with $A_2^* \sim (A^*, \varphi_2^* A_2)$ gives

$$A_2^* \sim (\varphi_2^* S_1^*, A^*) \tag{6.114}$$

Let $\varphi^* = \varphi_2^* \circ \varphi_1^*$, $B^* = \varphi_2^* B_1^*$, $O^* = (A^*, B^*)$, and $\Phi^* = (N_1^*)(\varphi_2^* S_1^*, \Sigma_2^*)$. Let us prove that the diagram

$$\begin{tikzcd} & O_1^* \arrow[dl, "\varphi_2^*"'] \arrow[dr, "\psi_1"] & \\ O^* \arrow[rr, "\psi_2"] & & O' \end{tikzcd} \tag{6.115}$$

commutes (compare with (6.112)). Since $A_1^* \trianglelefteq A_2$ and $A^* = \text{Below}(A_2^*, \varphi_2^* A_1^*)$ and $O_2 \xrightarrow{\varphi_2^*} O_2^*$ we have $O_1^* \xrightarrow{\varphi_2^*} O^*$, by Lemma 6.1.3. This shows that the left-hand side morphism exists. But then, since $O_1^* \xrightarrow{\psi_1} O'$ and (6.112) commutes, we have that the bottom morphism exists and that (6.115) commutes (the fact that O^* covers O' on N of B

[2]This is the logical flaw; we do not get by induction that $A_2^* = \text{Below}(A_0, \varphi_2^* A_2)$, for some A_0. Had we formulated our theorem as a completeness property of algorithm W we would know that A_2^* is of this form by definition of $W_{prinsigexp}$.

follows from Lemma 5.1.3). By composing the diagrams (6.109) and (6.115) we get the desired $O \xrightarrow{\varphi^*} O^*$.

Let us show that O^* is grounded. This requires that A^* is grounded in N of $\varphi_2^* B_1^*$; but this holds since A_2^* is grounded in N of $\varphi_2^* B_1^* \cup N_1^*$: let (θ, CE) occur in A^* not within a functor signature. Then it occurs free, i.e., tynames $\theta \subseteq$ names A^*. Then (θ, CE) occurs in A_2^* by definition of A^*. As A_2^* is grounded either θ is a type name (and then groundedness holds) or tynames $\theta \subseteq N$ of $\varphi_2^* B_1^* \cup N_1^*$, but as tynames $\theta \subseteq$ names A^* and names $A^* \cap N_1^* = \emptyset$ we have tynames $\theta \subseteq N$ of $\varphi_2^* B_1^*$. Thus A^* is grounded in N of $\varphi_2^* B_1^*$. Groundedness of O^* also requires that $\varphi_2^* B_1^*$ is grounded: let (θ, CE) occur in $\varphi_2^* B_1^*$ not within a functor signature and assume θ is not a type name (if it is $\varphi^* B_1^*$ is grounded). First assume that θ occurs free. Then as $A^* \sqsupseteq \varphi_2^* B_1^*$ and A^* is grounded we have tynames $\theta \subseteq N$ of $\varphi_2^* B_1^*$ as desired. Second assume θ occurs bound. Then by definition of application of a realisation (name capture is avoided) $(\theta, CE) = \varphi_2^*(\theta', CE')$ for some θ' and CE', and (θ', CE') occurs bound in B_1^* and θ' is not a type name. Then some type name $t \in$ tynames θ' is bound, and as we are allowed to rename bound names, we can assume that $t \notin N$ of $Bstar_1$. But then, as B_1^* is grounded, θ' has to be a type name, contradicting our assumption. Thus we conlude that O^* is grounded.

By Theorem 5.2.1 on (6.108) we get

$$O^* \vdash sigexp_1 \Rightarrow \varphi_2^*((N_1^*)S_1^*) \tag{6.116}$$

Also, since φ_2^* is fixed on N_1^* and $\Sigma_1^* \sqsubseteq A_1^*$ and $\varphi_2^* A_1^* \sqsubseteq A^*$ and (6.113) we have $\varphi_2^*((N_1^*)S_1^*) = (N_1^*)(\varphi_2^* S_1^*)$. Thus (6.116) reads $O^* \vdash sigexp_1 \Rightarrow (N_1^*)\varphi_2^* S_1^*$, i.e.

$$A^*, \varphi_2^* B_1^* \vdash sigexp_1 \Rightarrow (N_1^*)\varphi_2^* S_1^* \tag{6.117}$$

Expanding (6.111) we get

$$A_2^*, \varphi_2^* B_1^* + N_1^* + \{strid \mapsto \ \varphi_2^* S_1^*\} \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{6.118}$$

Thus by Lemma 6.2.2 on (6.118) and (6.114) we have

$$(\varphi_2^* S_1^*, A^*), \varphi_2^* B_1^* + N_1^* + \{strid \mapsto \varphi_2^* S_1^*\} \vdash sigexp_2 \Rightarrow \Sigma_2^* \tag{6.119}$$

We now wish to use rule 4.27 on (6.117) and (6.119). The side-condition $N_1^* \cap \mathrm{names}(A^*) = \emptyset$ is met by (6.113). Moreover, Φ^* is well-formed, for the following reasons. $(N_1^*)(\varphi_2^* S_1^*)$ is well-formed by (6.117), Σ_2^* is well-formed by (6.118) and if (m, E) is a structure occurring in Σ_2^* with $m \notin N_1^*$ then since $A_2^* \sqsupseteq \Sigma_2^*$ and (6.114) we have $m \in$ 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 wellformed and that $A^* \sqsupseteq \Phi^*$. Thus rule 4.27 applies and we get the desired $O^* \vdash \textit{funsigexp} \Rightarrow \Phi^*$.

Finally, let $\psi = \psi_2$. We wish to prove that the diagram

$$\begin{tikzcd} & O \arrow[dl, "\varphi^*"'] \arrow[dr, "\varphi"] & \\ O^* \arrow[rr, "\psi"] & & O' \end{tikzcd} \tag{6.120}$$

commutes and that $\psi\Phi^* = \Phi'$. But (6.120) commutes since (6.109) and (6.115) commute. Moreover, as $A^* \sqsupseteq \Phi^*$ and $O^* \xrightarrow{\psi} O'$ and $\text{names}\, O^* \cap N_1^* = \text{names}\, O' \cap N_1^* = \emptyset$, 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

$$\begin{aligned} \psi(\Phi^*) &= (N_1^*)(\psi\varphi_2^* S_1^*, \psi\Sigma_2^*) & \\ &= (N_1^*)(\psi_1 S_1^*, \Sigma_2') & \text{as (6.112) commutes} \\ &= (N_1')(S_1', \Sigma_2') & \text{by (6.110)} \\ &= \Phi' & \end{aligned}$$

as required. □

Using the above theorem we can finally prove the existence of principal signatures.

Theorem 6.2.2 (Principal Signatures) *Let B be a basis, A an assembly and assume (A, B) is grounded and let $A, B \vdash \textit{sigexp} \Rightarrow S$, for some S. Then there exists a principal signature for sigexp in (A, B).*

Proof Assume $A, B \vdash \textit{sigexp} \Rightarrow S$. Let $O = (A, B)$. Then $O \xrightarrow{\text{Id}} O$. Thus by Theorem 6.2.1 there exist O^*, φ^* and S^* depending on *phrase* and O only, such that $O \xrightarrow{\varphi^*} O^*$ and $O^* \vdash \textit{sigexp} \Rightarrow S^*$. Moreover, there exits a ψ such that the diagram

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

commutes and $\psi S^* = S'$. Let $(A^*, B^*) = O^*$, let $A_0^* = \text{Below}(A^*, \varphi^* A)$, $O_0^* = (A_0^*, B^*)$ and $\Sigma^* = \text{Clos}_{A_0^*} S^*$. Then by Lemma 6.2.9 we have that Σ^* is principal for *sigexp* in O_0^*.

Thus $O_0^* \vdash sigexp \Rightarrow \Sigma^*$. We have $O \xrightarrow{\varphi^*} O_0^*$, by Lemma 6.1.3. Then, since the above diagram commutes, the diagram

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

commutes. But from $O_0^* \xrightarrow{\psi} O$ and $O_0^* \vdash sigexp \Rightarrow \Sigma^*$ we get $O \vdash sigexp \Rightarrow \psi(\Sigma^*)$ by Theorem 5.2.1. In particular, $\psi\Sigma^*$ is principal for *sigexp* in O. $\square$

Chapter 7

Inferring Principal Signatures

In this chapter we give an algorithm which computes the principal signature for a signature expressions if the signature expression elaborates at all. It is based directly on the proof of Theorem 6.2.1. In the second section of this chapter we describe how to detect illegal signature expressions.

7.1 Algorithm W

The proof of Theorem 6.2.1 is constructive in the following sense. Given a grounded object $O \in \mathrm{Obj}$, a signature expression *sigexp* and the knowledge that there exists some O' and S' such that $O \xrightarrow{\mathrm{Id}} O'$ and $O' \vdash sigexp \Rightarrow S'$, then the proof gives a construction of O^* and Σ^* such that $O \xrightarrow{\mathrm{Id}} O^*$ and Σ^* is principal of *sigexp* in O^* (see Theorem 6.2.2).

This construction can be expressed as an algorithm W_{sigexp}. Notice that the Theorem 6.2.1 corresponds closely to the completeness result of Damas' and Milner's algorithm W [4]. In summa, the completeness of the algorithm W_{sigexp} follows from Theorems 6.2.1 and 6.2.2 (as discussed on page 54 we shold have done it the other way round and proved the existence of principal elaborations by proving completeness of W_{sigexp}). We have not proved that W_{sigexp} fails, if the signature expression cannot be elaborated at all, but in the next section we describe how to detect illegal signature expressions.

Our algorithm appears in Figures 7.1–7.13. It consists of a set of mutually recursive functions, one for each group of inference rules in Section 4.13 (we do not show the functions for elaborating types and type-expression rows — the rules in the Definition are deterministic, so there should be no confusion of what to do.) In the figures, we use the following notation. For all $O = (A, B) = (N, F, G, E)$, we write

$Admissify(O, longstrid_1, \ldots, longstrid_n)$ to mean $Admissify(A', N)$, where

$$A' = \Big((m, \{X \mapsto B(longstrid_1)\}), \ldots, ((m, \{X \mapsto B(longstrid_n)\}), A)\Big)$$

where m is a fresh structure name and X is an arbitrary structure identifier. $Admissify(A', N)$ is to be the algorithm corresponding to the proof of Theorem 5.1.1 — note that the proof of this theorem is also constructive (it should be obvious how the algorithm corresonding to this proof is, so it is not included.) Id denotes the identity realisation.

Let us, as an example, see how W_{spec} in the case of sequential specifications is obtained from the proof of Theorem 6.2.1. (The proof-case occurs on Page 71 and the algorithm in Figure 7.4.) The first recursive call in the algorithm corresponds to the induction on $spec_1$ and O (and other objects). The second recursive call corresponds to the induction on $spec_1$ and $O_2 = (A_1^*, B_1^* + E_1^*)$ in the proof. Finally, the returned values correspond to the constructed objects O^*, φ^* and E^* in the proof.

Next, les us consider the case of functor signature expressions. (The proof-case occurs on Page 77, and the algorithm in Figure 7.2.) The two recursive calls and the construction of O^*, φ^*, and Φ^* in the algorihthm corresponds closely to the proof, as above. What about the test then? This test is used since we in the proof rely on the existence of O', φ and Φ' to prove $N_1^* \cap \mathrm{names}\, A^* \neq \emptyset$ (when we apply Lemma 6.2.10.) This is analogous to the admissification theorem in the Commentary (Page 102–103), where the *construction* of the principal admissifier φ^* does not depend on the existence of an admissifier, but the proof of φ^* being well-defined and an admissifier hinges on the existence of an admissifier, c.f. Exercise 10.4 in the Commentary. Therefore, in an implementation of admissification one has to check that the constructed admissifier really is an admissifier; and likewise we have to test explicitly that $N_1^* \cap \mathrm{names}\, A^* \neq \emptyset$ in our algorithm. An example of a signature expression which attempts to violate this side-condition is shown below (taken from [16]).

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

$W_{sigexp}(O \text{ as } (A, B), sigexp) : \mathrm{Obj} \times \mathrm{Rea} \times \mathrm{Str} =$
case $sigexp$ of
`sig` $spec$ `end` => (* rule 4.1 *)
let $(O_1^* \text{ as } (A_1^*, B_1^*), \varphi^*, E^*) = W_{spec}(O, spec)$
$S^* = (m^*, E^*)$, where m^* is fresh
$O^* = ((S^*, A_1^*), B_1^*)$
in (O^*, φ^*, S^*)
| $sigid$ => (* rule 4.2 *)
let $(N^*)S^* = B(sigid)$, where all names in N^* are chosen to be fresh
$O^* = ((S^*, A), B)$
in (O^*, Id, S^*)

$W_{prinsigexp}(O \text{ as } (A, B), sigexp) : \mathrm{Obj} \times \mathrm{Rea} \times \mathrm{Sig} =$ (* rule 4.3 *)
let $(O^* \text{ as } (A^*, B^*), \varphi^*, S^*) = W_{sigexp}(O, sigexp)$
$A_0^* = \mathrm{Below}(A^*, \varphi^* A)$
$\Sigma^* = \mathrm{Clos}_{A_0^*} S^*$
in $((A_0^*, B^*), \varphi^*, \Sigma^*)$

Figure 7.1: Algorithms W_{sigexp} and $W_{prinsigexp}$

In [15] each name was ascribed a rank (a natural number) which was used to check the side-condition. This is unnecessary as explained above. In [16] the side-condition is checked the same way as here; the author, however, discovered independently of Tofte, that one can avoid the use of ranks.

The test in algorithm W_{exdesc} concerning exception descriptions, corresponding to rule 4.21, is needed because we in the proof case for structural contractions (recall that rule 4.21 is a structural contraction) rely on the existence of O', φ and Q'. Specifically, it is when we collect the results of the two applications of induction and prove that the side-conditions of the rule under consideration are satisfied (c.f. the wording "and were satisfied at (6.18)" on Page 59.)

Algorithms $W_{datdesc}$ and $W_{condesc}$ have been constructed on the basis of the proof-case for datatype specifications, and algorithm $W_{typdesc}$ has been constructed on the basis of the proof-cases for type and equality type specifications.

$W_{funsigexp}(O \text{ as } (A, B), funsigexp) : \text{Obj} \times \text{Rea} \times \text{FunSig} = (\text{* rule 4.27 *})$
$\quad \text{let}\,(strid : sigexp_1) : sigexp_2 = funsigexp$
$\qquad (O_1^* \text{ as } (A_1^*, B_1^*), \varphi_1^*, \Sigma_1^*) = W_{prinsigexp}(O, sigexp_1)$
$\qquad (N_1^*)S_1^* = \Sigma_1^*, \text{ where all names in } N_1^* \text{ are chosen to be fresh}$
$\qquad O_2 = (S_1^*, A_1^*), B_1^* + N_1^* + \{strid \mapsto S_1^*\}$
$\qquad (O_2^*, \varphi_2^*, \Sigma_2^*) = W_{prinsigexp}(O_2, sigexp_2)$
$\qquad A_2^* = A \text{ of } O_2^*$
$\qquad A^* = \text{Below}(A_2^*, \varphi_2^*(A_1^*))$
$\qquad \varphi^* = \varphi_2^* \circ \varphi_1^*$
$\qquad \Phi^* = (N_1^*)(\varphi_2^* S_1^*, \Sigma_2^*)$
$\quad \text{in if } N_1^* \cap \text{names } A^* \neq \emptyset \text{ then } \textbf{fail}$
$\qquad \text{else } ((A^*, \varphi_2^* B_1^*), \varphi^*, \Phi^*)$

Figure 7.2: Algorithm $W_{funsigexp}$

7.2 Detection of Illegal Signature Expressions

In this section we describe how to detect illegal signature expressions (signature expressions that do not elaborate) — to be practically useful in a compiler the inference algorithm of course should be able to detect illegal signature expressions.

The admissification algorithm can fail. In the Commentary it is explained in Section 10.5 and in the answer to exercise 10.4 how one detects that no admissifier exists in Standard ML. As explained in Section 5.1 our definition of of admissifier is slightly more restrictive than the definition used in SML. As can be seen from the proof of Theorem 5.1.1, the existence of a principal admissifier now also (compared to SML) relies on the existence of an admissifier when one has to prove the cover property of the constructed admissifier. Therefore, we add the following to the list in the answer to Exercise 10.4 in the Commentary

- In the absence of φ we can no longer apply lemma 5.1.2 to conclude that A covers $\varphi^* A$ on N, which we will therefore have to check.

Now there is only two more places where the algorithm can fail, namely in $W_{funsigexp}$ where it is checked that the side-condition on rule 4.27 is satisfied, and in W_{exdesc} where it is checked that the side-condition on rule 4.21 is satisfied.

$W_{spec}(O \text{ as } (A, B), spec) : \mathrm{Obj} \times \mathrm{Rea} \times \mathrm{Env} =$
case *spec* of
`val` *valdesc* => (* rule 4.4 *)
let $(O^*, VE^*) = W_{valdesc}(O, valdesc)$
$E^* = \mathrm{Clos}VE^*$ in Env
in (O^*, Id, E^*)
| `type` *typdesc* => (* rule 4.5 *)
let $(O^*, TE^*) = W_{typdesc}(O, typdesc, \mathrm{false})$
$E^* = TE^*$ in Env
in (O^*, Id, E^*)
| `eqtype` *typdesc* => (* rule 4.6 *)
let $(O^*, TE^*) = W_{typdesc}(O, typdesc, \mathrm{true})$
$E^* = TE^*$ in Env
in (O^*, Id, E^*)
| `datatype` *datdesc* => (* rule 4.7 *)
let $(O^*, (VE^*, TE^*)) = W_{datdesc}(O, datdesc)$
$E^* = (VE^*, TE^*)$ in Env
in (O^*, Id, E^*)
| `exception` *exdesc* => (* rule 4.8 *)
let $(O^*, EE^*) = W_{exdesc}(O, exdesc)$
$E^* = (EE^*, EE^*)$ in Env
in (O^*, Id, E^*)

Figure 7.3: Algorithm W_{spec}

| `structure` *strdesc* => (* rule 4.9 *)

let $(O^*, \varphi^*, SE^*) = W_{strdesc}(O, strdesc)$

$E^* = SE^*$ in Env v

in (O^*, φ^*, E^*)

| `functor` *fundesc* => (* rule 4.10 *)

let $(O^*, \varphi^*, F^*) = W_{fundesc}(O, fundesc)$

$E^* = F^*$ in Env

in (O^*, φ^*, E^*)

| `sharing` *shareq* => (* rule 4.11 *)

let $(O^*, \varphi^*) = W_{shareq}(O, shareq)$

$E^* = \{\}$ in Env

in (O^*, φ^*, E^*)

| `local` $spec_1$ `in` $spec_2$ => (* rule 4.12 *)

let $(O_1^* \text{ as } (A_1^*, B_1^*), \varphi_1^*, E_1^*) = W_{spec}(O, spec_1)$

$(O_2^*, \varphi_2^*, E_2^*) = W_{spec}((A_1^*, B_1^* + E_1^*), spec_2)$

$O^* = (A \text{ of } O_2^*, \varphi_2^* B_1^*)$

in $(O^*, \varphi_2^* \circ \varphi_1^*, E_2^*)$

| `open` $longstrid_1 \ \ldots \ longstrid_n$ => (* rule 4.13 *)

let $E_i = E \text{ of } B(longstrid_i), \quad \text{for } 1 \leq i \leq n$

$E^* = E_1 \ + \ \cdots \ + E_n$

in (O, Id, E^*)

| `include` $sigid_1 \ \ldots \ sigid_n$ => (* rule 4.14 *)

let $E_i = E \text{ of Instance}(B(longstrid_i)), \quad \text{for } 1 \leq i \leq n$

$E^* = E_1 \ + \ \cdots \ + E_n$

in (O, Id, E^*)

| *empty* => $(O, \mathrm{Id}, \{\})$ (* rule 4.15 *)

| $spec_1\langle;\rangle spec_2$ => (* rule 4.16 *)

let $(O_1^* \text{ as } (A_1^*, B_1^*), \varphi_1^*, E_1^*) = W_{spec}(O, spec_1)$

$(O_2^*, \varphi_2^*, E_2^*) = W_{spec}((A_1^*, B_1^* + E_1^*), spec_2)$

in $((A \text{ of } O_2^*, \varphi_2^* B_1^*), \varphi_2^* \circ \varphi_1^*, \varphi_2^* E_1^* + E_2^*)$

Figure 7.4: Algorithm W_{spec} — continued

$W_{valdesc}(O \text{ as } (A, B), valdesc) : \mathrm{Obj} \times VE = $ (* rule 4.17 *)
case *valdesc* of
$\quad var : \ ty =>$
$\qquad \text{let } \tau = W_{ty}(C \text{ of } B, ty)$
$\qquad \text{in } (O, \{var \mapsto \tau\})$
$\ | \ var : \ ty$ `and` $valdesc =>$
$\qquad \text{let } \tau = W_{ty}(C \text{ of } B, ty)$
$\qquad\quad VE = VE \text{ of } W_{valdesc}(O, valdesc)$
$\qquad \text{in } (O, \{var \mapsto \tau\} + VE)$

Figure 7.5: Algorithm $W_{valdesc}$

$W_{typdesc}(O \text{ as } (A, B), typdesc, eq) : \mathrm{Obj} \times TE = $ (* rule 4.18 *)
case *typdesc* of
$\quad tyvarseq\ tycon \ =>$
$\qquad \text{let } k = |\ tyvarseq\ |$
$\qquad\quad \tau^* = $ a fresh type name with arity k and equality eq
$\qquad\quad TE^* = \{tycon \mapsto (\tau^*, \{\})\}$
$\qquad \text{in } (((TE^*, A), B), TE^*)$
$\ | \ tyvarseq\ tycon$ `and` $typdesc \ =>$
$\qquad \text{let } k = |\ tyvarseq\ |$
$\qquad\quad \tau^* = $ a fresh type name with arity k and equality eq
$\qquad\quad TE = TE \text{ of } W_{typdesc}(O, typdesc, eq)$
$\qquad\quad TE^* = \{tycon \mapsto (\tau^*, \{\})\} + TE$
$\qquad \text{in } (((TE^*, A), B), TE^*)$

Figure 7.6: Algorithm $W_{typdesc}$

$W_{datdesc}(O \text{ as } (A, B), datdesc) : \mathrm{Obj} \times (VE \times TE) =$ (* rule 4.19 *)
  case $datdesc$ of
    $tyvarseq_1\ tycon_1 = condesc_1$ `and`...`and` $tyvarseq_n\ tycon_n\ = condesc_n$ =>
      let $k_i = \mid tyvarseq_i \mid$
        $t_i^* =$ a fresh type name with arity k_i not admitting equality
        $\tau_i = tyvarseq_i\ \tau_i^*$
        $CE_i^* = W_{condesc}(O, \tau_i, condesc_i)$
        $VE^* = \mathrm{Clos}CE_1^* + \cdots + \mathrm{Clos}CE_n^*$
        $TE^* = \{tycon_1 \mapsto (t_1^*, \mathrm{Clos}CE_1^*)\} + \cdots + \{tycon_n \mapsto (t_n^*, \mathrm{Clos}CE_n^*)\}$
      in $(((TE^*, A), B), (VE^*, TE^*))$

Figure 7.7: Algorithm $W_{datdesc}$

$W_{condesc}(O \text{ as } (A, B), \tau, condesc) : CE =$ (* rule 4.20 *)
  case $condesc$ of
    con => $\{con \mapsto \tau\}$
  | con `of` ty =>
      let $\tau^* = W_{ty}(C \text{ of } B, ty)$
      in $\{con \mapsto \tau^* \to \tau\}$
  | con `|` $condesc$ =>
      let $CE = CE$ of $W_{condesc}(O, condesc)$
      in $\{con \mapsto \tau\} + CE$
  | con `of` ty | $condesc$ =>
      let $\tau^* = W_{ty}(C \text{ of } B, ty)$
        $CE = CE$ of $W_{condesc}(O, condesc)$
      in $\{con \mapsto \tau^* \to \tau\} + CE$

Figure 7.8: Algorithm $W_{condesc}$

$W_{exdesc}(O \text{ as } (A, B), exdesc) : \text{Obj} \times EE = (^* \text{ rule } 4.21\ ^*)$
case *exdesc* of
$\quad excon$ => $(O, \{excon \mapsto \texttt{exn}\})$
$\quad | \; excon$ `of` ty =>
$\qquad$ let $\tau^* = W_{ty}(C \text{ of } B, ty)$
$\qquad$ in if $\text{tyvars}(ty) \neq \emptyset$ then **fail**
$\qquad\quad$ else $(O, \{excon \mapsto \tau^* \to \texttt{exn}\})$
$\quad | \; excon$ `and` $exdesc$ => v
$\qquad$ let $EE = EE \text{ of } W_{exdesc}(O, exdesc)$
$\qquad$ in $(O, \{excon \mapsto \texttt{exn}\} + EE)$
$\quad | \; excon$ `of` ty `and` $exdesc$ =>
$\qquad$ let $\tau^* = W_{ty}(C \text{ of } B, ty)$
$\qquad\quad EE = EE \text{ of } W_{exdesc}(O, exdesc)$
$\qquad$ in if $\text{tyvars}(ty) \neq \emptyset$ then **fail**
$\qquad\quad$ else $(O, \{excon \mapsto \tau^* \to \texttt{exn}\} + EE)$

Figure 7.9: Algorithm W_{exdesc}

$W_{strdesc}(O \text{ as } (A, B), strdesc) : \text{Obj} \times \varphi \times SE = (^* \text{ rule } 4.22\ ^*)$
case *strdesc* of
$\quad strid : sigexp$ =>
$\qquad$ let $(O^*, \varphi^*, S^*) = W_{sigexp}(O, sigexp)$
$\qquad$ in $(O^*, \varphi^*, \{strid \mapsto S^*\})$
$\quad | \; strid : sigexp$ `and` $strdesc$ =>
$\qquad$ let $(O_1^*, \varphi_1^*, S^*) = W_{sigexp}(O, sigexp)$
$\qquad\quad (O_2^*, \varphi_2^*, SE) = W_{strdesc}(O_1^*, strdesc)$
$\qquad$ in $(O_2^*, \varphi_2^* \circ \varphi_1^*, \{strid \mapsto \varphi_2^* S^*\} + SE)$

Figure 7.10: Algorithm $W_{strdesc}$

$W_{fundesc}(O \text{ as } (A, B), fundesc) : \text{Obj} \times \varphi \times \text{F} =$ (* rule 4.23 *)
  case *fundesc* of
    funid funsigexp =>
      let $(O^*, \varphi^*, \Phi^*) = W_{funsigexp}(O, funsigexp)$
      in $(O^*, \varphi^*, \{funid \mapsto \Phi^*\})$
  | *funid funsigexp* `and` *fundesc* =>
      let $(O_1^*, \varphi_1^*, \Phi^*) = W_{funsigexp}(O, funsigexp)$
        $(O_2^*, \varphi_2^*, F) = W_{fundesc}(O_1^*, fundesc_1)$
      in $(O_2^*, \varphi_2^* \circ \varphi_1^*, \{funid \mapsto \varphi_2^* \Phi^*\} + F)$

Figure 7.11: Algorithm $W_{fundesc}$

$W_{shareq}(O \text{ as } (A, B), shareq) : \text{Obj} \times \varphi =$
  case *shareq* of
    $longstrid_1 = \cdots = longstrid_n$ => (* rule 4.24 *)
      let $\varphi^* = Admissify(O, longstrid_1, \ldots, longstrid_n)$
      in $(\varphi^* O, \varphi^*)$
  | $longtycon_1 = \cdots = longtycon_n$ => (* rule 4.25 *)
      let $\varphi^* = Admissify(O, longtycon_1, \ldots, longtycon_n)$
      in $(\varphi^* O, \varphi^*)$
  | $shareq_1$ `and` $shareq_2$ => (* rule 4.26 *)
      let $(O_1^*, \varphi_1^*) = W_{shareq}(O, shareq_1)$
        $(O_2^*, \varphi_2^*) = W_{shareq}(O_1^*, shareq_2)$
      in $(O_2^*, \varphi_2^* \circ \varphi_1^*)$

Figure 7.12: Algorithm W_{shareq}

$W_{sigbind}(O, sigbind) : G =$ (* rule 4.28 *)
  case *sigbind* of
    sigid `=` *sigexp* =>
      let $(O^*, \varphi^*, \Sigma^*) = W_{prinsigexp}(O, sigexp)$
      in if TypeExplicit(Σ^*) then
          case EqualityPrincipal(Σ^*) of
            OK(Σ_1^*) => $\{sigid \mapsto \Sigma_1^*\}$
          | FAIL => **fail**
        else **fail**
  | *sigid* `=` *sigexp* `and` *sigbind* =>
      let $(O^*, \varphi^*, \Sigma^*) = W_{prinsigexp}(O, sigexp)$
        $G = W_{sigbind}(B, sigbind)$
      in if TypeExplicit(Σ^*) then
          case EqualityPrincipal(Σ^*) of
            OK(Σ_1^*) => $\{sigid \mapsto \Sigma_1^*\} + G$
          | FAIL => **fail**
        else **fail**

Figure 7.13: Algorithm $W_{sigbind}$

Chapter 8

Implementation of W in the ML Kit

The ML Kit [3] is an implementation of Standard ML written in Standard ML, which follows the Definition very closely. It is basically an interpreted system and has been written without any considerations to efficiency.

I have implemented algorithm W and *Admissify* including detection of illegal signature expressions in the ML Kit. This has been done by modifying and replacing some parts of the existing implementation of Modules Elaboration in the ML Kit (I have also modified the parser as the grammar has changed; this will not be described here.) In this chapter we describe the implementation — the description is based on a chapter, written by the author, describing Modules Elaboration in the ML Kit [3]. We do not assume that the reader is familiar with the ML Kit.

The implementation has not been tested rigorously, but in Appendix A we present some sample runs, which shows that the implementation seems to be capable of infererring principal signatures and to detect illegal signature expressions. The examples include the most interesting examples presented in the chapters of this report.

We have followed the style of Kit programming in the sense that the implementation follows the semantics presented earlier on very closely. Therefore our description is brief; we will only try to supply just enough information that it becomes possible to navigate around the code — this should hopefully make it easier for people who would like to modify or extend our implementation in some way.

The outline of the chapter is as follows. Section 8.1 gives an overview of the Elaborator. Sections 8.2–8.14 describe the implementation of the semantic objects and operations. The constituent files of the Modules Elaborator are described in Section 8.15.

The order of the subsections of the present Chapter follows the order of the subsections of Chapter 4.

8.1 Elaboration

Algorithm W is implemented by a functor which returns a structure which can be constrained by the following signature.

```
signature ELABTOPDEC =
  sig
    type StaticBasis

    type PreElabTopdec and PostElabTopdec

    val elab_topdec: StaticBasis * PreElabTopdec ->
                         StaticBasis * PostElabTopdec
    type StringTree
    val layoutStaticBasis: StaticBasis -> StringTree
  end;
```

The elaborator takes an abstract syntax tree as argument (of type `PreElabTopdec`) and a static basis, and returns a new static basis and a new abstract syntax tree, which is decorated with error information if the argument phrase does not elaborate.

The implementation of algorithm W is found in functor `ElabTopdec`. Below we show the skeleton of the functor and a piece of code corresponding to a part of W_{spec} (and corresponding to rule 4.16).

```
functor ElabTopdec(structure IG: TOPDEC_GRAMMAR
    structure OG: TOPDEC_GRAMMAR
    structure ElabDec: ELABDEC
    structure ModuleStatObject: MODULE_STATOBJECT
    structure ModuleEnvironments: MODULE_ENVIRONMENTS
    structure U : MODULE_UNIFY
                 ....
               ) : ELABTOPDEC  =
  struct
```

```
  ....

  fun elab_spec (O : Env.Obj, spec : IG.spec) :
     (Env.Obj * Stat.Realisation * Env.Env * OG.spec) =

    case spec of

      (* sequential specifications *)
      IG.SEQspec(i, spec1, spec2) =>
let
  val (O1, rea1, E1, out_spec1) = elab_spec(O, spec1)
        val A1 = Env.A_of_Obj O1
        val B1 = Env.B_of_Obj O1
        val (O2, rea2, E2, out_spec2) =
            elab_spec(Env.mkObj(A1, B1 B_plus_E E1), spec2)
        val O' = Env.mkObj(Env.A_of_Obj O2, rea2 onB B1)
in
  (O', rea2 oo rea1, (rea2 onE E1) E_plus_E E2,
   OG.SEQspec(conv i, out_spec1, out_spec2))
end

  ....
end;
```

Notice that functor `ElabTopdec` is parameterised on the objects of the static semantics of the Modules (structures matching signature `MODULE_STATOBJECT` and signature `MODULE_ENVIRONMENTS` and on a module `U` implementing admissification.

8.2 Semantic Objects

The specification of the semantic objects for the Modules and the operations on these objects are given by the following signatures.

Signature	*Specifies*
`MODULE_STATOBJECT`	Simple semantic objects and operations in the static semantics
`MODULE_ENVIRONMENTS`	Environments and operations on these

Notice that the implementation of the semantic objects has been divided into two, as the implementation of environments need not know the exact implementation of the underlying semantic objects.

The functors `ModuleStatObject` and `ModuleEnvironments` implement the semantic objects. They are both parameterised on the modules implementing the semantic objects for the Standard ML Core Language, as specified by signatures `STATOBJECT_PROP` and `ENVIRONMENTS_PROP`. Moreover, part of the implementation of the semantic objects resides in functor `Environments` which returns a structure matching `ENVIRONMENTS_PROP` since environments contain structure environments and are needed in Core Elaboration.

The implementation of the semantic objects, except assemblies (the implementation of which is described below), is straight out of the Definition and of Section 4.1.

Recall from Chapter 7 that assemblies are needed during admissification, and that we need the operations Below, Clos and names. By inspection of the W-functions in the preceding chapter, we see that when the Clos operation is used, it is on an assembly returned by the Below operation. Moreover, when the operation names is used on an assembly it is also on an assembly returned by the Below operation. So functor-components are trivial when the names and Clos operations are used. Moreover, there are no requirements on functor components in admissification, except for the domain requirements in the definition of cover. Hence, it is sufficient to represent skel(m, E) for every (m, E) in an assembly.

We use the following types for assemblies, defined in functor `ModuleEnvironments`.

```
type offspring_Str  = (strid, StrName) FinMap.map
 and offspring_Ty   = (tycon, TyStr  ) FinMap.map
 and offspring_Doms = {Fdom   : funid EqSet.Set}

type offspring      = {structures : offspring_Str,
                       types : offspring_Ty,
                       doms : offspring_Doms};

type Assembly_Str   = (StrName, offspring)FinMap.map
```

```
 and Assembly_Ty    = (TypeFcn, TyStr)FinMap.map
type Assembly       = {m_arcs : Assembly_Str,
                       theta_arcs : Assembly_Ty}
```

Notice that we for each structure record the domains of the functor environments of the structure's environment — at first sight it seems perhaps as though we have have left out the domains of the structure and type environments, but these are represented by the domains of the mappings `structures` and `types` in an `offspring`.

Assemblies are also represented in the existing ML Kit implementation, although assemblies are not manipulated directly in the inference rules in Standard ML. They are used to check for cover, see Commentary Chapter 11, and to check for consistency, see Commentary Section 10.5. The only change compared to the existing implementation is that we have added `doms` to the `offspring` type. The reason is simply that our definition of cover is different compared to the definition of cover used in SML [12, Section 5.13]. The domains are needed to check for cover during admissification, c.f. Section 7.2.

Notice also that the representation is "flat", in the sense that `offspring_Str` has `StrName` as its range type (instead of `offspring`). Consider an assembly consisting of the structure

```
structure S0 =
  struct
    val x = 1
    structure S10 =
      struct
        structure S20 = struct end
      end
    structure S11 =
      struct
      end
  end
```

The representation of the assembly is then

```
{ m_arcs = { m0  ↦ { structures = { S10 ↦ m10,
                                    S11 ↦ m11 },
                     types =      {} },
```

```
                 doms =      { Fdom =  {} }
         m10 ↦ { structures = { S20 ↦ m20 },
                 types =     {} },
                 doms =      { Fdom =  {} }
         m11 ↦ { structures = {},
                 types =     {} },
                 doms =      { Fdom =  {} }
         m20 ↦ { structures = {},
                 types =     {} },
                 doms =      { Fdom =  {} }
theta_args = {} }
```

where we as usual use `{}` around records and $\{\}$ around finite maps and sets. This flat representation is chosen because it makes the implementation of admissification easier.

Also notice that for a type structure θ in the domain of `theta_arcs`, θ is mapped to (θ, CE) for some CE. That is, θ is represented twice. (Actually, there is no good reason for this.)

Functor `ModuleEnvironments` implements operations to build assemblies from other semantic objects; we must be able to turn a basis into an assembly when we start elaborating a signature binding or signature expression (recall that SML module phrases are elaborated against a basis, so we have to turn the basis into an assembly somewhere.) The functor also implements functions satisfying the following specifications (from signature `MODULE_ENVIRONMENTS`)

```
val Alookup_Str         : Assembly * StrName -> offspring_Str
and Aoffspring_Str_Fold : (((strid * StrName) * 'b) -> 'b) -> 'b ->
                           offspring_Str -> 'b
val Alookup_Ty          : Assembly *  StrName -> offspring_Ty
and Aoffspring_Ty_Fold  : (((tycon * TyStr) * 'b) -> 'b) -> 'b ->
                          offspring_Ty -> 'b
val Alookup_TypeFcn     : Assembly * TypeFcn -> TyStr
```

The `_Fold` functions are simply functions which fold a function over the elements in the range of a finite map, and the `Alookup_` functions are, of course, lookup functions. Other operations on assemblies are also implemented; some of them are described in the relevant sections below.

8.3 Consistency

As explained in the Section 7.2 consistency need only be checked during admissification. Admissification is implemented by functor `ModuleUnify`; the implementation of the necessary check for consistency of type structures [12, Section 5.2], function `CheckCEdom`, is in this functor.

8.4 Well-formedness

It suffices to check well-formedness of type structures [12, Section 4.9]. This is to be done during admissification (Commentary Section 10.5 and solution to Exercise 10.4). The implementation of the check occurs in functor `ModuleUnify` (function `CheckTyStrWF`).

8.5 Cycle-freedom

As explained in Section 7.2 it suffices to check for cycle-freedom during admissification. Function `cyclic` in functor `ModuleEnvironments` takes an assembly as argument and returns true iff the assembly is cyclic. The cycle test is performed by a mindless depth-first search (for each structure name).

8.6 Cover

Recall from Section 7.2 that cover, must be checked by *Admissify*. Function `covers`, implemented in functor `ModuleEnvironments`, takes a name set, and two assemblies A and A^* and checks whether or not A covers A^* on the name set. (A^* is supposed to be φA where φ is the realisation constructed during admissifiction.)

The implementation follows the definition of cover, Section 4.5, closely.

8.7 Admissiblity

Admissibility, see Section 4.6, is implemented by separate checks for consistency, well-formedness and cycle-freedom as described above. No admissibility predicate is provided or needed.

8.8 Type Realisation

Type realisations are implemented by functor `ModuleStatObject`. Actually, the operations are provided by one of the argument structures to `ModuleStatObject`. The argument structure matches signature `STATOBJECT_PROP`, which in turn specifies part of the static objects and operations for the Core. Type realisations are only needed in the static semantics for the Modules, but the implementation of type realisation needs to know the representation of type names and type functions. Therefore, type realisations are implemented in functor `StatObject`.

8.9 Realisation

A realisation, [12, Section 5.7], is implemented in functor `ModuleStatObject` simply as a record consisting of a type realisation (see above) and a structure realisation (implemented by a SML function). Operations for applying realisations to the representations of semantic objects are also implemented here.

8.10 Type Explication

Check for type explication is implemented by a function `type_explicit`, specified in signature `MODULE_STATOBJECT` by

```
val type_explicit : Sig -> bool
```

and implemented in functor `ModuleStatObject`.

Given a signature $(N)S$, the function first checks the signatures occuring inside the signature (in functor signatures) recursively. If they are type-explicit, the function simply collects the set of explicit type names in S (without considering the functor signatures), and compares this set with the set of bound names, and returns true if they are equal.

8.11 Signature Instantiation

Signature instantiation is not needed to implement elaboration of signature expressions or signature bindings so it is not described.

8.12 Below

Below is implemented straightforwardly by fixed point iteration as suggested by the characterization of below in Section 6.1.

8.13 Equality-Principal Signature

Equality principality is implemented in functor `ModuleStatObject` by function `equality_principal`, which takes a signature $(N)S$ as argument. Equality-principality is implemented almost straight out of the definition of equality-principality. Given a signature, `equality_principal` first maximises equality and then checks that the signature respects equality. However, as we have signatures inside signatures (in functor signatures) we have to be a bit careful with the order in which the maximisation is done. Note that, as remarked in Section 3.4, the equality attributes of type names in a functor signature Φ occuring in a signature $(N)S$ has no significance for the equality attributes on the rest of the type names in the signature $(N)S$. For a signature $(N)S$, the implementation first maximises equality with respect to the bound datatype names in N; then for each functor signature $(N_1)(S_1,(N_1')S_1')$ in turn, $(N_1)S_1$ is first maximised and then $(N_1')S_1'$ is maximised. We first maximise with respect to the bound datatype names in N, as the equality attribute of a type name in a functor can depend on the equality attribute of one of the names in N.

Example 2 Consider the following signature expression.

```
sig
  datatype t = C
  functor F(X: sig datatype t' = D of t end) : sig end
end;
```

The equality attribute for `t'` depends on the equality attribute for `t`. □

For a functor signature $(N_1)(S_1,(N_1')S_1')$, we first maximise with respect to N_1 as the equality attribute of type names in N_1' can depend on the equality attribute of the type names in N_1.

Example 3 Consider the following functor specification.

```
functor F(X: sig datatype t = C end) :
  sig datatype t' = D of t end;
```

The equality attribute for `t'` depends on the equality attribute for `t`. □

8.14 Admissify

In this section we briefly describe the admissification algorithm, *Admissify*.

Admissification is implemented in functor `ModuleUnify` by the functions `unifyTy` and `unifyStr`, which are specified in signature `MODULE_UNIFY`:

```
datatype UnifyResult =
    OK of Realisation
  | ERROR of ErrorInfo
val unifyStr: NameSet * Assembly *
              (StrName * longstrid) list -> UnifyResult
and unifyTy : NameSet * Assembly *
              (TyStr   * longtycon) list -> UnifyResult
```

If the admissification succeeds, it returns the most general admissifier.

For a structure sharing specification we look up the structures bound to the structure identifiers in the sharing specification, and call `unifyStr` with a list of the names of the structures. Actually, as one sees from the type of `unifyStr`, each structure name is paired with the structure identifier via which it was found — this is for error reporting only.

For a type sharing specification we similarly look up the type structures bound to the type constructors and call `unifyTy`.

The implementation of `unifyTy` and `unifyStr` follows the proof of the Admissification Theorem and the notes given in Section 7.2 and in the solution to Exercise 10.4 in the Commentary . As the code is well-commented and follows the descriptions very closely, we will give only one little remark here. In `unifyTy` we replace the constructor environments in the argument type structures by constructor environments looked up in the assembly. The reason can be seen from the following example. Consider the elaboration of the second sharing specification in the following signature expression

```
sig
```

```
    datatype t1 = C of int
    type t2
    sharing type t1 = t2
    datatype t3 = D of int
    sharing type t2 = t3
  end
```

If we simply look up the constructor environment for `t2` in the basis we get an empty constructor environment, ignoring that `t2` shares with `t1` and the second sharing would (wrongly) appear to be legal. But since the Kit looks the type structures up in the assembly we get the non-empty constructor environment (with domain {`C`}) and so the Kit spots that the sharing specification violates the third condition of consistency (conflicting non-empty constructor environment domains.)

8.15 Files

The relevant files are listed below. They all reside in directory `Common`.

File	Contains
`MODULE_STATOBJECT.sml`	Signature for the semantic objects
`MODULE_ENVIRONMENTS.sml`	Signature for the environments
`MODULE_UNIFY.sml`	Signature for the admissication process
`ModuleStatObject.sml`	Semantic objects
`ModuleEnvironments.sml`	The environments of the semantic objects
`ModuleUnify.sml`	The admissification process
`ElabTopdec.sml`	The Modules Elaborator corresponding to the inference rules

Chapter 9

Conclusion and Future Work

In this chapter we briefly conclude on the work presented in this paper, and describe possible future work.

9.1 Conclusion

We have presented an extension of the Standard ML Modules Language in which specification of functors inside signatures is allowed, based on Tofte's work on HML [15] [16].

We have defined the static semantics of signature expressions and signature bindings and proven that if a signature expression elaborates at all, then it elaborates to a principal signature. We have found out, however, that the proof formally contains a logical flaw; we have deferred making the straightforward but time consuming correction to future work. Moreover, we have presented an algorithm for inferring principal signatures — the completeness of the algorithm follows from the proof of principal elaborations (as discussed it should actually be the other way round, principal elaborations should be proved by proving completeness of the algorithm) — and described how to detect illegal signature expressions. The algorithm including detection of illegal signature expressions has been implemented in the ML Kit; we have given an overview of the implementation and shown sample runs.

9.2 Future Work

The existence of principal signatures is, as mentioned in the Introduction chapter, important because if principal signatures did not exist, one would have to re-elaborate the functor body for each functor application. However, in the case of higher-order functor application it seems that there will be an overhead for every functor application even though principal signatures exist as we have proved. The problem is the propagation of sharing. In the first-order language the propagation of sharing from the argument of a functor to the result of the application can be computed once and for all when the functor is declared. It is represented by a common name-prefix — recall that functor signatures are of the form $(N)(S, (N')S')$, bound names (in N) which occur free in both S and S' represent the propagation of sharing. In the higher-order language, the propagation of sharing depends on the functor components in the actual argument to the application, and therefore cannot be computed once and for all.

Example 4 As an example, consider the following piece of code (where we have used the derived form for functors).

```
functor F(functor G(X': sig end): sig end
          structure Y: sig end) =
   G(Y);

functor Id(X: sig end) = X;

structure S  = struct end;

structure Z  = F(functor G = Id, structure Y = S);

structure Z' = Id(S);
```

(The structure `S` is arbitrary.) In SML, there is sharing between `S` and `Z'`, and we would expect it to be the case that `S` and `Z` also shares (that `Z` and `Z'` are identical). However, this is only because `F` is applied to `Id`; if we had applied `F` to another functor whose argument and result did not share, then `Z` and `Z'` should not share. □

Mads Tofte and David MacQueen have given a set of inference rules for higher-order functor application where they formalise the propagation of sharing [17]. I do not yet fully understand the properties of this semantics but hope to investigate it in the future. As it seems a bit complicated, it could also be interesting to investigate the possibility of using a record typing discipline for higher-order program modules. Aponte [1] [2] have given a formulation of the SML Modules Language based on record typing which could be used as a basis.

In this paper we have not considered dynamic semantics at all, but to complete the work on higher-order functors one should of course also define a dynamic semantics. We foresee no problems with this. Given the dynamic semantics one should try to prove consistency between the static and dynamic semantics (that "well-typed modules do not go wrong"). Aponte has already shown consistency for her formalisation of the Standard ML Modules Language [1].

Epilogue

It has been a great experience to work on this project; I feel that I have learned a lot. When I began this project, I knew almost nothing about the Standard ML Modules Language; I had only read the chapter in Paulson's book [13] and had almost no programming experience with modules. Also, my experience with SML was modest. Then the project began by reading the Definition and Commentary, and then later [15] and [16]. To begin with the idea was not to prove the existence of principal signatures for the full higher-order modules language but merely to implement the elaboration algorithm in the ML Kit, but as I was interested in understanding all the details the balance of the project shifted.

All in all, I believe this present report is a good effort when one takes into consideration the background for writing the report and the time available for the project.

Appendix A

Sample Runs

The examples all have the same form: first comes the input source program (where we in comments have written what result we expect) and then comes the resulting output. The output is obtained by setting some debugging flags in the ML Kit (that's the reason why sometimes a signature is shown for a signature expression that does not elaborate — the debugging information is just printed, exept for the case where the error handling is horrible and an exception Crash is raised).

```
(*
 * t1.sml
 *
 * Cover example --- from Section on Cover in Chapter Discussion
 *
 * Expected result: Elaborates, X,X',X'' all shares, names of D components
 * in X' and X'' equal.
 *)

signature SIG_1 =
sig
  structure X  : sig end
  structure X' :
    sig
      structure D : sig end
    end
  functor F(X0:
            sig
              structure X'' :
                sig
                  structure D : sig end
                end
              sharing X'' = X
```

```
          end): sig end
  sharing X = X'
end;

(* Result:


- elabFile "../testprogs/t1.sml";
**SUCCESS**
signature
   SIG_1 =
   sig
      structure X : sig  end; structure X' : sig structure D : sig  end end;
      functor
         F
         (X0:
            sig
               structure X'' : sig structure D : sig  end end;
               sharing X'' = X
            end):
         sig  end;
      sharing X = X'
   end
<iBas: >
{{}}

signature SIG_1 :
   ({{}, 4, 5, 11})
   sig(11)
      functor F
         (({{}, 8})
          sig(8)
             structure X'' : sig(5)  structure D : sig(4)      end    end
          end
         ) : ({{}, 10}) sig(10)      end
      structure X' : sig(5)  structure D : sig(4)      end    end
      structure X : sig(5)      end
   end

> signature SIG_1 =
    sig
      structure X :
        sig
        end
      structure X' :
        sig
          structure D :
```

```
            sig
            end
        end
    end
**EOF**
val it = () : unit
*)

(*
 * t2.sml
 *
 * Cover example --- from Section on Cover in Chapter Discussion
 *
 * Expected result: Does not elaborate, no cover.
 *
 *)

structure Empty = struct end;
signature SIG_2 =
  sig
    local
      structure NonEmpty :
sig
  structure Dangle : sig end
end
      sharing Empty = NonEmpty
    in
    end
  end;

(* Result:


- elabFile "../testprogs/t2.sml";
**SUCCESS**
structure Empty = struct  end
<iBas: >
{{}, 15}    structure Empty : sig(15)      end
> structure Empty :
    sig
    end
**SUCCESS**
No cover --- no structure cover
signature
   SIG_2 =
   sig
      local
```

```
         structure NonEmpty : sig structure Dangle : sig  end end;
         sharing Empty = NonEmpty
      in
      end
   end
<iBas: >
{{}}  signature SIG_2 : ({{}, 18}) sig(18)      end
> signature SIG_2 =
    sig
    end
**EOF**
val it = () : unit

*)

(*
 * t3.sml
 *
 * Cover example --- from Section on Cover in Chapter Discussion
 *
 * Expected result: Does not elaborate, no cover, conflicting
 * constructor domains.
 *
 *
 *)

datatype t = C of int;
signature SIG_3 =
  sig
    local
      datatype t' = C of int | D of int
      sharing type t = t'
    in
    end
  end;

(* Result:

- elabFile "../testprogs/t3.sml";
**SUCCESS**
datatype t = C of int
<iBas: >
{{int(E 1), t(E 16)}}

t LAMBDA (). t(E 16)/ C (int(E 1) -> t(E 16))
con C: (int(E 1) -> t(E 16))
```

```
> datatype t
    con C : (int(E 1) -> t(E 16))
**SUCCESS**
Conflicting constructor domains
signature
   SIG_3 =
   sig
      local datatype t' = C of int | D of int; sharing type t = t' in  end
   end
<iBas: >
{{}}  signature SIG_3 : ({{}, 19}) sig(19)      end
> signature SIG_3 =
    sig
    end
**EOF**
val it = () : unit

*)



(*
 * t4.sml
 *
 * Principal signatures and well-formedness. From section on
 * Principal signatures in Chapter Discussion.
 *
 * Expected result: Elaborates, the name of S3 should be quantified in the
 * argument signature for F1.
 *
 *)

signature SIG_4 =
  sig
    structure S0 : sig end
    functor F1(X:
       sig
 structure S1 : sig end
 structure S2 :
   sig
     functor F2(Y:
sig
  structure S3 : sig end
  sharing S3 = S1
end): sig end
   end
 sharing S2 = S0
 structure S1 : sig end
```

```
       end): sig end
  end;

(* Result:


- elabFile "../testprogs/t4.sml";
**SUCCESS**
signature
   SIG_4 =
   sig
      structure S0 : sig  end;
      functor
         F1
         (X:
            sig
               structure S1 : sig  end;
               structure
                  S2
                  :
                  sig
                     functor
                        F2
                        (Y:
                           sig
                              structure S3 : sig  end; sharing S3 = S1
                           end):
                        sig  end
                  end;
               sharing S2 = S0;
               structure S1 : sig  end
            end):
         sig  end
   end
<iBas: >
{{}}

signature SIG_4 :
   ({{}, 20, 33})
   sig(33)
      functor F1
         (({{}, 28, 27, 21})
          sig(28)
             structure S2 :
                sig(20)
                   functor F2
                      (({{}, 23})
```

```
                       sig(23)  structure S3 : sig(21)      end    end
                      ) : ({{}, 25}) sig(25)      end
                end
             structure S1 : sig(27)      end
          end
         ) : ({{}, 32}) sig(32)      end
      structure S0 : sig(20)      end
   end

> signature SIG_4 =
    sig
      structure S0 :
        sig
        end
    end
**EOF**
val it = () : unit
*)


(*
 * t5.sml
 *
 * Cover / Flexible / Rigid structures --- only possible to add components
 * to flexible structures. Section on Flexible and Rigid structures in Chapter
 * Discussion
 *
 * Expected result: Does not elaborate.
 *
 *)

structure Empty = struct  end;
signature NONEMPTY =
  sig
    structure NonEmpty :
      sig
structure Dangle : sig end
      end
    sharing NonEmpty = Empty
  end;


(* Result:

- elabFile "../testprogs/t5.sml";
**SUCCESS**
structure Empty = struct  end
```

```
<iBas: >
{{}, 34}    structure Empty : sig(34)      end
> structure Empty :
    sig
    end
**SUCCESS**
No cover --- no structure cover
signature
   NONEMPTY =
   sig
      structure NonEmpty : sig structure Dangle : sig  end end;
      sharing NonEmpty = Empty
   end
<iBas: >
{{}}

signature NONEMPTY :
   ({{}, 35, 36, 37})
   sig(37)
      structure NonEmpty :
         sig(36)  structure Dangle : sig(35)      end    end
   end

> signature NONEMPTY =
    sig
      structure NonEmpty :
        sig
          structure Dangle :
            sig
            end
        end
    end
**EOF**
val it = () : unit

*)


(*
 * t6.sml
 *
 * Side-condition on rule for functor signature expressions.
 * Section on Algorithm W in Chapter Inferring Principal Signatures.
 *
 * Expected result: Does not elaborate.
 *
 *)
```

```
signature SIG_6 =
sig
  structure A : sig end
  functor F (X: sig end):
    sig
      sharing X = A
    end
end;

(* Result:

- elabFile "../testprogs/t6.sml";
**SUCCESS**
Impossible: Attempt to violate side-condition on rule funsigexp

uncaught exception Crash

*)
```

Bibliography

[1] Maria Virginia Aponte. *Typage d'un Système de Modules Paramétriques avec Partage: Une Application de l'unification dans les Théories Equationelles.* PhD thesis, Université de Paris 7, Feb. 1992.

[2] Maria Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 465–478. ACM, Jan. 1993.

[3] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit, Version 1. Distributed with the ML Kit, March 1993.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[5] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS–LFCS–86–2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland, March 1986.

[6] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[7] Xavier Leroy. Polymorphic typing of an algorithmic language. Technical Report 1778, INRIA, Domaine de Voluceau, Rocquencourt, B.P.105, 78153 le Chesnay Cedex, France, Oct 1992. English version of the author's PhD-thesis.

[8] David MacQueen. Modules for Standard ML. In *Conf. Rec. of the 1984 ACM Symp. on LISP and Functional Programming*, pages 198–207, Aug. 1984.

[9] David MacQueen. Modules for Standard ML. Technical report, AT & T Bell Laboratories, October 1985. PART III of [5].

[10] Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[11] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[13] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[14] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 77–88. ACM, Jan. 1989.

[15] Mads Tofte. Principal signatures for higher-order program modules. In *Proc. Principles of Programming Languages (POPL)*, pages 189–199, Jan. 1992.

[16] Mads Tofte. Principal signatures for higher-order program modules. To appear in Journal of Functional Programming, 1993.

[17] Mads Tofte and David B. MacQueen. Higher-order functor application. Working Note, April 1992.