A DENOTATIONAL SEMANTICS FOR ML MODULES

*** Draft 25 November 1985 ***

Don Sannella
University of Edinburgh

1 An introduction to ML modules

blah blah ... blah etc. including signature/structure closure

2 Restrictions and extensions

Some restrictions to ML have been adopted in formulating the semantics which appears below. Only the purely applicative subset of ML is treated (so assignment and exceptions are not permitted). Furthermore, we do not allow use of polymorphic types or transparent type bindings with complex right-hand sides (e.g. **type t = t1 * t2** is forbidden while **type t = t1** is allowed), and we do not allow a type and value with the same name to be present in the same context. Assignment is forbidden for foundational reasons, although it would probably be possible to extend the foundations to permit it. The other restrictions are purely for the sake of simplicity. The semantics of type abstraction has not yet been included, and the effects of infix and nonfix directives has been relegated to the level of concrete syntax.

Some of the syntactic forms listed in the next section are not present in MacQueen's proposal or are mentioned there as possible extensions but not required. They have been included because adding them does not further complicate the semantics. The order of specifications in a signature is (intentionally) not taken to be significant in signature matching. Types and values are treated uniformly with respect to sharing, which is an extension to MacQueen's proposal.

3 Syntax

PROGRAMS *prog*

prog ::= **signature** $sigb_1$ **and** ... **and** $sigb_n$ $\quad n \geq 1$
functor $funb_1$ **and** ... **and** $funb_n$ $\quad n \geq 1$
dec
prog {;} *prog'*

SIGNATURE BINDINGS *sigb*

sigb ::= *atid* = *sig*

FUNCTOR BINDINGS *funb*

funb ::= *atid*(*plist*) {: *sig*} = *str*

plist ::= $atid_1$: sig_1, ..., $atid_n$: sig_n {sharing $patheq_1$ and ... $patheq_m$} $n \geq 0,\ m \geq 1$

patheq ::= id_1 = ... = id_n $n \geq 1$

STRUCTURE BINDINGS *strb*

strb ::= *atid* {: *sig*} = *str*

SIGNATURES *sig*

sig ::= *atid*
    sig *spec* **end**

spec ::= **val** $atid_1$: ty_1 **and** ... **and** $atid_n$: ty_n $n \geq 1$
    type *spectb*
    datatype *db*
    spec {;} *spec'*
    structure $specstrb_1$ **and** ... **and** $specstrb_n$ {**sharing** $patheq_1$ **and** ... **and** $patheq_m$}[1] $n \geq 1$, $m \geq 1$

spectb ::= *atid*
    atid = *id*[2]
    spectb **and** *spectb'*

specstrb ::= *atid* : *sig*

STRUCTURES *str*

str ::= *id*
    struct *declist* **end**
    str : *sig*[3]
    atid(str_1,...,str_n) $n \geq 0$

---

[1] MacQueen's proposal does not require that sharing constraints be allowed except in functor parameter specifications, but it is not more difficult to allow them in signatures as well.

[2] Type identities in signatures are not required by MacQueen's proposal, but they are easy to provide.

[3] This form does not appear in MacQueen's proposal, but it is an easy and natural addition and seems to be in the spirit of Standard ML.

DECLARATIONS *dec*

dec ::= `val` vb
`type` tb
`datatype` db
`local` $declist$ `in` $declist'$ `end`
`open` $id_1 \ldots id_n$ $\quad n \geq 1$
`structure` $strb_1$ `and` ... `and` $strb_n$ $\quad n \geq 1$

$declist$ ::= dec
$declist$ {;} $declist'$

TYPE BINDINGS *tb* and DATATYPE BINDINGS *db*

tb ::= $atid$ = id
tb `and` tb'

db ::= $atid_1$ = $constrs_1$ `and` ... `and` $atid_n$ = $constrs_n$ $\quad n \geq 1$

The syntax of value bindings (vb) is just as it is in the Standard ML core language, except that only an *atid* (not an *id*) is allowed in a binding occurrence on the left-hand side of a value binding. *Atid* denotes an atomic identifier (a sequence of alphanumeric characters) while *id* denotes a normal ML identifier, possibly containing dots: id ::= $atid$ | $atid.id$.

4 Values

A *sigval* (the denotation of a signature) is a 4-tuple $\mathfrak{Sig} = \langle \xi, N, \tau, \Sigma \rangle$ where:

- $Substrs[\mathfrak{Sig}] =_{def} \xi$ is a map $id \longmapsto id$
- $Names[\mathfrak{Sig}] =_{def} N$ is a set of identifiers
- $AlgSig[\mathfrak{Sig}] =_{def} \Sigma$ is an *algebraic signature*, i.e. a set of type names T and a set of value names V where each $v \in V$ is assigned a *type* built from the type names in T and the type constructors $\rightarrow$ and $*$; $Names[\Sigma] =_{def} T \cup V$
- $\tau : N \rightarrow Names[\Sigma]$ is a function

A *strval* (the denotation of a structure) is a 6-tuple $\mathfrak{Str} = \langle tag, \xi, N, \tau, \Sigma, C \rangle$ where $Sig[\mathfrak{Str}] =_{def} \langle \xi, N, \tau, \Sigma \rangle$ is a sigval, $Tag[\mathfrak{Str}] =_{def} tag$ is an identifier and $Alg[\mathfrak{Str}] =_{def} C$ is a Σ-algebra[4] associating interpretations to the type and value names of Σ. Think of C as the code associated with the types and values of Σ (in practice, only values have code associated with them; types are merely conceptually associated with sets of values). Sometimes it is convenient to write a strval as a ($id \times sigval \times algebra$)-triple instead of as a 6-tuple. For any sigval $\mathfrak{Sig}$, the class of all strvals $\mathfrak{Str}$ with $Sig[\mathfrak{Str}] = \mathfrak{Sig}$ is denoted $Str[\mathfrak{Sig}]$.

---

[4] Our use of the terms *algebraic signature* and *algebra* is somewhat loose, since we are dealing with higher-order types and partial functions. This misuse will not lead to any problems since no use is made of any results from algebra.

MacQueen's proposal gives closure rules which signatures and structures are required to satisfy. These rules are also satisfied by every sigval and strval. But although the signature closure rule indirectly constrains structures by requiring that each one has a signature satisfying that rule (this means in effect that every type used must have a name belonging to the structure), not all strvals satisfy such a constraint. This is because strvals are used to denote partially-elaborated structures, which are not required to satisfy this constraint, as well as fully-elaborated structures.

Primitive-str is a strval containing the pervasive type and value bindings described in section 5 of the Standard ML core language proposal (**bool**, **int**, **true**, etc.). These types and values are automatically a part of every structure and every signature.

The sigval/strval component ξ and the strval component *tag* are needed to deal with structure sharing and the generative aspect of structure declaration, respectively. Since each elaboration of an encapsulated structure declaration or functor application creates a distinct structure, each strval must carry a distinct tag to distinguish it from strvals which are identical but created separately. This also provides a means by which strvals which are really the same can be identified. The component ξ gives the correspondence between substructure names and their tags; if a structure A in the structure environment has a tag t then **struct** ... **structure** B = A ... **end** creates a strval (with its own tag) where the substructure name B is associated with the tag t. Thus in a strval, ξ indicates the extent to which substructures share with each other and with external structures and substructures. In a sigval, ξ only reflects internal substructure sharing since sharing with external structures/substructures is not possible.

The sigval/strval components N and τ are needed to deal with sharing of types and values. N contains names which can be used in programs to refer to types and values, Σ contains the unique internal names of these types and values (associated in a strval with interpretations by C), and τ gives the correspondence between names in N and the internal names in Σ. Several names in N may correspond to the same internal name in Σ, and if an internal name in the algebraic signature component of a strval appears in the algebraic signature component of another strval then it denotes there the same type or value; again, in a sigval sharing with types/values belonging to other sigvals cannot occur (except with pervasive types/values). Complex type synonyms (e.g. **type t = t1 * t2**) are forbidden in order to simplify the semantics. Such synonyms could be allowed by making τ map names to types built from the type names in Σ and the type constructors -> and *. A sigval or strval includes types and values defined at "top level" within the corresponding signature/structure as well as types and values belonging to substructures. Types and values belonging to a substructure A have names of the form A.n; furthermore, every type or value having a name of this form is regarded as a part of A.

This is not the only possible way of representing signature and structure values, although any alternative representation must take proper account of the complications mentioned above (generative structure declarations, structure sharing and multiple names for a single type or value). Other possible representations are discussed below under "On the representation of signature and structure values".

A *funval* (the denotation of a functor) is a 6-tuple $\mathfrak{Fun} = \langle params, \mathfrak{Sig}_{par}, str, \rho, \psi, \pi \rangle$ where:

- *params* is an *atid*-list (the formal parameter names)
- $\mathfrak{Sig}_{par}$ (the combined formal parameters with sharing taken into account) is a sigval as above
- *str* is a structure expression (the body of the functor, qualified by the result signature if any)
- ρ, ψ, π are the structure, signature and functor environments at the point of declaration

The structure and signature environments ρ and ψ are maps $atid \longmapsto strval$ and $atid \longmapsto sigval$ respectively. The functor environment π is a map $atid \longmapsto funval$. The structure environment includes bindings of structures occurring earlier than the construct currently being elaborated, as well as (if the current construct is a structure) bindings of its substructures. The latter is necessary because in a nested context a substructure of the current structure is just like a previously-defined structure.

5 Semantic operations

Convention: A operation of type $\ldots \times strval \times \ldots \to strval$ gives rise to a operation with the same name of type $\ldots \times sigval \times \ldots \to sigval$. by simply ignoring the tag and algebra components of the strval argument(s). There is one exception to this rule: the operation *addsubstrs*, for which something slightly more complex must be done — see below.

Notation: When Σ and Σ' are algebraic signatures and $\Sigma \subseteq \Sigma'$, $\iota_{\Sigma \subseteq \Sigma'}$ denotes the inclusion. This notation is also used when $\iota_{\Sigma \subseteq \Sigma'}$ is an injection rather than an inclusion (as is the case when the construction of Σ' involves tagging: $\Sigma' = \ldots \cup tag(\Sigma)$).

5.1 Restricting to a subset of the names in a strval

$restrict(\mathfrak{Str}, N')$ is that sub-strval $\mathfrak{Str}'$ of $\mathfrak{Str}$ such that $Names[\mathfrak{Str}'] = N'$. Note that $\mathfrak{Str}'$ may contain anonymous types even if $\mathfrak{Str}$ does not. The result is regarded as the same structure for purposes of sharing iff the restriction is trivial (no names are forgotten). A substructure in $\mathfrak{Str}$ is in the result if any of its type or value names are retained, and it is regarded as a "new" substructure iff some of its type or value names are forgotten.

restrict : *strval* × *id-set* $\dot\rightarrow$ *strval*

$restrict(\langle tag,\xi,N,\tau,\Sigma,C\rangle,N') =$

$$\text{let tag}' = \begin{cases} tag & \text{if } N=N' \\ \text{a new tag} & \text{otherwise} \end{cases} \text{ in}$$

$$\begin{aligned}\text{let } \xi' = \{(id\mapsto t)\in\xi \mid\ & \exists n\in N'.\ n \text{ is of the form } id.m \\ & \text{and } \forall n\in N.\ n \text{ is of the form } id.m \text{ implies } n\in N'\} \\ \cup\ \{id\mapsto tag(id) \mid\ & id\in dom(\xi),\ tag(id) \text{ is a (different) new tag for each } id, \\ & \exists n\in N'.\ n \text{ is of the form } id.m \\ & \text{and } \exists n\in N.\ n \text{ is of the form } id.m \text{ and } n\notin N'\} \text{ in}\end{aligned}$$

let Σ' = the smallest subsignature of Σ containing $\tau(N')$ in

$\langle\ tag',\ \xi',\ N',\ \tau\restriction N',\ \Sigma',\ C|_{\Sigma'}\ \rangle$

<u>error</u> if $N'\not\subseteq N$

5.2 Fitting a strval to a sigval

fit(𝔖𝔱𝔯,𝔖𝔦𝔤) checks if the candidate strval 𝔖𝔱𝔯 matches the target sigval 𝔖𝔦𝔤; if it does then the strval 𝔖𝔱𝔯' which results from restricting 𝔖𝔱𝔯 to fit 𝔖𝔦𝔤 is returned. The result of *fit* is guaranteed to be signature-closed (this is a consequence of the fourth error check below). The third error check might be relatively expensive to implement in practice, since it involves examining every pair of names in 𝔖𝔦𝔤. However, if two structures A and B share then every pair of names A.n, B.n shares. This means that if the fifth error check succeeds then some of the pairs of names in 𝔖𝔦𝔤 need not be checked.

fit : *strval* × *sigval* → *strval*

$fit(\langle tag,\xi,N,\tau,\Sigma,C\rangle,\langle\xi',N',\tau',\Sigma'\rangle) = restrict(\langle tag,\xi,N,\tau,\Sigma,C\rangle,N')$

<u>error</u> if $N'\not\subseteq N$

or types/values in $\tau'(N')$ do not correspond with types/values in $\tau(N')$,
i.e. $\exists n\in N'.\ \tau'(n)\in Types[\Sigma'] \not\Leftrightarrow \tau(n)\in Types[\Sigma]$

or types/values in $\tau(N')$ do not share at least as much as types/values in $\tau'(N')$,
i.e. $\exists n,m\in N'.\ \tau'(n)=\tau'(m)$ and $\tau(n)\neq\tau(m)$

or ~~the types of values in $\tau'(N')$ do not correspond with the types of values in $\tau(N')$~~;
i.e. $\exists n\in N'.\ \tau'(n)\in Vals[\Sigma']$ and $\tau(\tau'^{-1}(type[\tau'(n)]))\neq type[\tau(n)]$

or substrs in the candidate do not share at least as much as those in the target,
i.e. $\exists id,id'\in dom(\xi').\ \xi'(id)=\xi'(id')$ and $\xi(id)\neq\xi(id')$

5.3 Checking if a strval has a closed signature

signature-closed(𝔖𝔱𝔯) yields *true* if 𝔖𝔱𝔯 satisfies the signature closure rule (i.e. if all types in *AlgSig*[𝔖𝔱𝔯] have names in *Names*[𝔖𝔱𝔯]) and *false* otherwise.

signature-closed : *strval* → *bool*

$$\text{signature-closed}\ \langle tag,\xi,N,\tau,\Sigma,C\rangle = \begin{cases} true & \text{if } \forall t\in Types[\Sigma].\ \exists id\in N.\ \tau(id) = t \\ false & \text{otherwise} \end{cases}$$

5.4 Combining strvals/environments

$\mathfrak{Str} \cup \mathfrak{Str}'$ is the union of $\mathfrak{Str}$ and $\mathfrak{Str}'$, containing all of the type and value bindings in $\mathfrak{Str}$ and $\mathfrak{Str}'$, which must not have type, value or substructure names in common. $\mathfrak{Str} + \mathfrak{Str}'$ adds the bindings of $\mathfrak{Str}'$ to those of $\mathfrak{Str}$, where bindings in $\mathfrak{Str}'$ may supercede those in $\mathfrak{Str}$. If $\mathfrak{Str}'$ contains a substructure with the same name as a substructure of $\mathfrak{Str}$, the entire substructure from $\mathfrak{Str}$ is replaced by the one in $\mathfrak{Str}'$ (not just the types and values with common names). Note that because each newly-declared type and value is assigned a unique internal name, $\mathfrak{Str}$ and $\mathfrak{Str}'$ are guaranteed not to conflict on the level of their algebra and algebraic signature components. Both $\cup$ and + produce a "new" strval. The analogous operation on maps $id \hookrightarrow \alpha$ (e.g. environments) is also provided.

$\cup$: *strval* × *strval* → *strval*

$\langle tag,\xi,N,\tau,\Sigma,C\rangle \cup \langle tag',\xi',N',\tau',\Sigma',C'\rangle =$
 let tag″ be a new tag in
 $\langle\ tag'',\ \xi \cup \xi',\ N \cup N',\ \tau \cup \tau',\ \Sigma \cup \Sigma',\ C \cup C'\ \rangle$
 <u>*error*</u> *if* $N \cap N' \neq \phi$
 or $dom(\xi) \cap dom(\xi') \neq \phi$

$+ : (id \hookrightarrow \alpha) \times (id \hookrightarrow \alpha) \rightarrow (id \hookrightarrow \alpha)$

$$(\delta + \delta')(\mathrm{id}) = \begin{cases} \delta'(id) & if\ id \in dom(\delta') \\ \delta(id) & if\ id \notin dom(\delta')\ and\ id \in dom(\delta) \\ undefined & otherwise \end{cases}$$

+ : *strval* × *strval* → *strval*

$\langle tag,\xi,N,\tau,\Sigma,C\rangle + \langle tag',\xi',N',\tau',\Sigma',C'\rangle =$
 let $N'' = \{id \in N \mid id$ *is not of the form* $id'.n$ *for some* $id' \in dom(\xi')\}$ *in*
 let $\tau'':N'' \cup N' \rightarrow Names[\Sigma \cup \Sigma']$ *be defined by*

$$\tau''(\mathrm{id}) = \begin{cases} \tau'(id) & if\ id \in dom(\tau') \\ \tau(id) & if\ id \notin dom(\tau')\ and\ id \in dom(\tau) \end{cases} \quad in$$

 let tag″ be a new tag in
 $restrict(\ \langle tag'',\xi+\xi',N'' \cup N',\tau'',\Sigma \cup \Sigma',C \cup C'\rangle,\ N'' \cup N'\ )$

5.5 Generating unique internal names for types/values

$tag(\mathfrak{Str},\Sigma')$ is the strval resulting from $\mathfrak{Str}$ by changing the internal names of types/values in $AlgSig[\mathfrak{Str}] - \Sigma'$ to make them distinct from all other internal type/value names (the function *tag*: *signature-fragment* → *signature-fragment* produces the new internal names by attaching uniquely-generated *tags* to them).[5] The result is a new strval, containing new

---

[5] Of course, *tag* is not a function since it produces different internal names each time it is called. It could be made into a function by passing to it a unique tag as an extra parameter. These tags could be taken from a list of tags passed as an extra parameter to the various functions which call *tag*. All these tags would originate from a list of tags passed to $\mathcal{Prog}$. These details have been suppressed because they would significantly clutter the semantic equations and hide more important details; see D. Sannella "A set-theoretic semantics for Clear", Acta Informatica 21, pp. 443-472 (1984) where in an analogous situation these details have not been suppressed.

substrvals (except for those with signatures wholly within Σ').

$tag : strval \times signature \rightarrow strval$

$tag(\langle tag,\xi,N,\tau,\Sigma,C\rangle,\Sigma') =$
$\quad let\ \Sigma'' = tag(\Sigma - \Sigma') \cup (\Sigma \cap \Sigma')\ in$
$\quad let\ tag'\ be\ a\ new\ tag\ in$
$\quad let\ \xi' = \{id \hookrightarrow tag(id) \mid id \in dom(\xi),\ \exists n \in N.\ n\ is\ of\ the\ form\ id.m\ and\ \tau(n) \notin \Sigma',$
$\qquad\qquad and\ tag(id)\ is\ a\ (different)\ new\ tag\ for\ each\ id\}$
$\qquad \cup \{id \hookrightarrow \xi(id) \mid id \in dom(\xi)\ and\ \forall n \in N.\ n\ is\ of\ the\ form\ id.m\ implies\ \tau(n) \in \Sigma'\}\ in$
$\quad \langle\ tag',\ \xi',\ N,\ \tau \circ \iota_{\Sigma \subseteq \Sigma''},\ \Sigma'',\ \iota_{\Sigma \subseteq \Sigma''}(C)\ \rangle$

5.6 Extracting a substructure from a strval

$substructure(id,\mathfrak{Str})$ is the strval corresponding to the substructure id of $\mathfrak{Str}$.

$substructure : id \times strval \rightarrow strval$

$substructure(id,\langle tag,\xi,N,\tau,\Sigma,C\rangle) =$
$\quad let\ \xi' = \{n \hookrightarrow t \mid (id.n \hookrightarrow t) \in \xi\}\ in$
$\quad let\ N' = \{n \mid id.n \in N\}\ in$
$\quad let\ \tau' = \{n \hookrightarrow id.n \mid id.n \in N\} \circ \tau\ in$
$\quad restrict(\ \langle \xi(id),\xi',N',\tau',\Sigma,C\rangle,\ N'\ )$

5.7 Adding new substructures to a strval

$addsubstrs(\{\langle atid_1,\mathfrak{Str}_1\rangle,\ldots,\langle atid_n,\mathfrak{Str}_n\rangle\},\mathfrak{Str}')$ is the strval which results from adding $\mathfrak{Str}_1,\ldots,\mathfrak{Str}_n$ to $\mathfrak{Str}'$ as substructures named $atid_1,\ldots,atid_n$ respectively. The result is a new strval.[6]

$addsubstrs : (atid \times strval)\text{-}set \times strval \rightarrow strval$

$addsubstrs(\{\langle atid_1,\langle tag_1,\xi_1,N_1,\tau_1,\Sigma_1,C_1\rangle\rangle,\ldots,\langle atid_n,\langle tag_n,\xi_n,N_n,\tau_n,\Sigma_n,C_n\rangle\rangle\},\mathfrak{Str}') =$
$\quad let\ tag'\ be\ a\ new\ tag\ in$
$\quad let\ \xi' = \{atid_1 \hookrightarrow tag_1,\ldots,atid_n \hookrightarrow tag_n\} \cup \{atid_1.id \hookrightarrow id \mid id \in dom(\xi_1)\} \circ \xi_1$
$\qquad\qquad \cup \ldots \cup$
$\qquad\qquad \{atid_n.id \hookrightarrow id \mid id \in dom(\xi_n)\} \circ \xi_n\ in$
$\quad let\ N' = \{atid_1.m \mid m \in N_1\} \cup \ldots \cup \{atid_n.m \mid m \in N_n\}\ in$
$\quad let\ \tau' = \{atid_1.m \hookrightarrow m \mid m \in N_1\} \circ \tau_1 \cup \ldots \cup \{atid_n.m \hookrightarrow m \mid m \in N_n\} \circ \tau_n\ in$
$\quad let\ \Sigma' = \Sigma_1 \cup \ldots \cup \Sigma_n\ in$
$\quad let\ C' = C_1 \cup \ldots \cup C_n\ in$
$\quad \langle\ tag',\ \xi',\ N',\ \tau',\ \Sigma',\ C'\ \rangle \cup \mathfrak{Str}'$

---

[6] The convention for viewing strval operations as operations on sigvals does not work in the case of *addsubstrs* because the *Substrs* component of the result depends on the *Tag* components of the strvals in the list which forms its first argument. To make it work for sigvals, it is necessary to regard every sigval in this list as tagged by a (different) uniquely generated tag.

5.8 Identifying types, values and substructures in a sigval

identify-type(*ta*,*tb*,$\mathfrak{Sig}$) is the sigval resulting from $\mathfrak{Sig}$ when the internal names of the types named *ta* and *tb* are identified. A new internal name is chosen for the type in the result, unless either *ta* or *tb* names a pervasive type. If both *ta* and *tb* name a pervasive type, then they must name the *same* type.

$identify\text{-}type : id \times id \times sigval \rightarrow sigval$

$identify\text{-}type(ta,tb,\langle \xi,N,\tau,\Sigma\rangle) =$

$$let\ internal = \begin{cases} \tau(ta) & if\ \tau(ta)\in Types[primitive\text{-}str] \\ \tau(tb) & if\ \tau(tb)\in Types[primitive\text{-}str] \\ a\ new\ tag & otherwise \end{cases}\quad in$$

$let\ \sigma = 1_{\tau(N)\ -\ \tau\{ta,tb\}} \cup \{\tau(ta)\hookrightarrow internal, \tau(tb)\hookrightarrow internal\}\ in$

$\langle\ \xi,\ N,\ \tau\circ\sigma,\ \sigma(\Sigma)\ \rangle$

<u>error</u> $if\ \tau(ta)\notin Types[\Sigma]\ or\ \tau(tb)\notin Types[\Sigma]$

$or\ \tau(ta)\in Types[primitive\text{-}str]\ and\ \tau(tb)\in Types[primitive\text{-}str]\ and\ \tau(ta)\neq\tau(tb)$

identify-value(*va*,*vb*,$\mathfrak{Sig}$) is the sigval resulting from $\mathfrak{Sig}$ when the internal names of the values named *va* and *vb* are identified. Note that *va* and *vb* must have the same type. A new internal name is chosen for the value in the result, unless either *va* or *vb* names a pervasive value. If both *va* and *vb* name a pervasive value, then they must name the *same* value.

$identify\text{-}value : id \times id \times sigval \rightarrow sigval$

$identify\text{-}value(va,vb,\langle \xi,N,\tau,\Sigma\rangle) =$

$$let\ internal = \begin{cases} \tau(va) & if\ \tau(va)\in Vals[primitive\text{-}str] \\ \tau(vb) & if\ \tau(vb)\in Vals[primitive\text{-}str] \\ a\ new\ tag & otherwise \end{cases}\quad in$$

$let\ \sigma = 1_{\tau(N)\ -\ \tau\{va,vb\}} \cup \{\tau(va)\hookrightarrow internal, \tau(vb)\hookrightarrow internal\}\ in$

$\langle\ \xi,\ N,\ \tau\circ\sigma,\ \sigma(\Sigma)\ \rangle$

<u>error</u> $if\ \tau(va)\notin Vals[\Sigma]\ or\ \tau(vb)\notin Vals[\Sigma]$

$or\ type[\tau(va)] \neq type[\tau(vb)]$

$or\ \tau(va)\in Vals[primitive\text{-}str]\ and\ \tau(vb)\in Vals[primitive\text{-}str]\ and\ \tau(va)\neq\tau(vb)$

identify-structure(*sa*,*sb*,$\mathfrak{Sig}$) is the sigval resulting from $\mathfrak{Sig}$ when the substructures named *sa* and *sb* are identified. A new tag is chosen for the substructure in the result. All of the corresponding types/values in the subtructures named *sa* and *sb* are also identified; the types must be identified first in case some of the values to be identified have types which include them.

$identify\text{-}structure : id \times id \times sigval \rightarrow sigval$

$identify\text{-}structure(sa,sb,\mathfrak{Sig}) =$
$\quad let\ Na = \{m \mid sa.m \in Names[\mathfrak{Sig}]\}$
$\quad and\ Nb = \{m \mid sb.m \in Names[\mathfrak{Sig}]\}\ in$
$\quad let\ \{\langle ta_1,tb_1\rangle,\ldots,\langle ta_p,tb_p\rangle\} = \{\langle sa.m,sb.m\rangle \mid m \in Na\ and\ \tau(sa.m),\tau(sb.m) \in Types[\mathfrak{Sig}]\}$
$\quad and\ \{\langle va_1,vb_1\rangle,\ldots,\langle va_q,vb_q\rangle\} = \{\langle sa.m,sb.m\rangle \mid m \in Na\ and\ \tau(sa.m),\tau(sb.m) \in Vals[\mathfrak{Sig}]\}\ in$
$\quad let\ \mathfrak{Sig}' = identify\text{-}type(ta_1,tb_1,\ldots,identify\text{-}type(ta_p,tb_p,\mathfrak{Sig})\ldots)\ in$
$\quad let\ \langle \xi'',N'',\tau'',\Sigma''\rangle = identify\text{-}value(va_1,vb_1,\ldots,identify\text{-}value(va_q,vb_q,\mathfrak{Sig}')\ldots)$
$\quad let\ strtag\ be\ a\ new\ tag\ in$
$\quad let\ \sigma = 1_{codom(\xi'') - \xi''\{sa,sb\}} \cup \{\xi''(sa) \hookrightarrow strtag, \xi''(sb) \hookrightarrow strtag\}\ in$
$\quad \langle\ \xi''\circ\sigma,\ N'',\ \tau'',\ \Sigma''\ \rangle$
$\ \underline{error}\ if\ Na \neq Nb$
$\qquad or\ \tau(sa.m) \in Types[\mathfrak{Sig}] \not\Leftrightarrow \tau(sb.m) \in Types[\mathfrak{Sig}]\ for\ some\ m \in Na$
$\qquad or\ sa \notin dom(\xi'')\ or\ sb \notin dom(\xi'')$

$identify(\{\langle a_1,b_1\rangle,\ldots,\langle a_n,b_n\rangle\},\mathfrak{Sig})$ is the sigval resulting from $\mathfrak{Sig}$ when the types/values/substructures named a_j and b_j are identified. Types are identified first, then substructures, and finally values.

$identify : (id \times id)\text{-}set \times sigval \rightarrow sigval$

$identify(\{\langle a_1,b_1\rangle,\ldots,\langle a_n,b_n\rangle\},\mathfrak{Sig}) =$
$\quad let\ \{\langle ta_1,tb_1\rangle,\ldots,\langle ta_p,tb_p\rangle\} = \{\langle a_j,b_j\rangle \mid a_j,b_j \in Names[\mathfrak{Sig}]\ and\ \tau(a_j),\tau(b_j) \in Types[\mathfrak{Sig}]\}$
$\quad and\ \{\langle sa_1,sb_1\rangle,\ldots,\langle sa_q,sb_q\rangle\} = \{\langle a_j,b_j\rangle \mid a_j,b_j \in dom(\xi)\}$
$\quad and\ \{\langle va_1,vb_1\rangle,\ldots,\langle va_r,vb_r\rangle\} = \{\langle a_j,b_j\rangle \mid a_j,b_j \in Names[\mathfrak{Sig}]\ and\ \tau(a_j),\tau(b_j) \in Vals[\mathfrak{Sig}]\}\ in$
$\quad let\ \mathfrak{Sig}' = identify\text{-}type(ta_1,tb_1,\ldots,identify\text{-}type(ta_p,tb_p,\mathfrak{Sig})\ldots)\ in$
$\quad let\ \mathfrak{Sig}'' = identify\text{-}structure(sa_1,sb_1,\ldots,identify\text{-}structure(sa_p,sb_p,\mathfrak{Sig}')\ldots)\ in$
$\quad identify\text{-}value(va_1,vb_1,\ldots,identify\text{-}value(va_p,vb_p,\mathfrak{Sig}'')\ldots)$
$\ \underline{error}\ if\ none\ or\ more\ than\ one\ of\ the\ following\ conditions\ are\ satisfied\ for\ some\ a_j,b_j:$
$\qquad -\ a_j,b_j \in Names[\mathfrak{Sig}]\ and\ \tau(a_j),\tau(b_j) \in Types[\mathfrak{Sig}]$
$\qquad -\ a_j,b_j \in dom(\xi)$
$\qquad -\ a_j,b_j \in Names[\mathfrak{Sig}]\ and\ \tau(a_j),\tau(b_j) \in Vals[\mathfrak{Sig}]$

5.9 Adding new type/value names

$joinnames(\mathfrak{Sig},newnames)$ is the sigval $\mathfrak{Sig}$ augmented by the names in *newnames*, each of which is provided with a distinct internal name. The names in *newnames* are required to be atomic, so none of the substructures of $\mathfrak{Sig}$ are altered.

$joinnames : sigval \times signature\text{-}fragment \rightarrow sigval$

$joinnames(\langle \xi,N,\tau,\Sigma\rangle,newnames) =$
$\quad let\ \Sigma' = \Sigma \cup tag(newnames)\ in$
$\quad \langle\ \xi,\ N \cup Names[newnames],\ \tau\circ\iota_{\Sigma\subseteq\Sigma'} \cup \iota_{newnames\subseteq\Sigma'},\ \Sigma'\ \rangle$
$\ \underline{error}\ if\ N \cap Names[newnames] \neq \phi$
$\qquad or\ any\ of\ the\ names\ in\ newnames\ is\ non\text{-}atomic$

$joinanon(\mathfrak{Sig},newnames)$ is the sigval $\mathfrak{Sig}$ augmented by the (internal) names in *newnames*;

external names are not provided.

$joinanon : sigval \times signature \rightarrow sigval$

$joinanon(\langle \xi, N, \tau, \Sigma \rangle, newnames) = \langle\ \xi,\ N,\ \tau,\ \Sigma \cup newnames\ \rangle$

$bind(\{\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle\}, \mathfrak{Str})$ is the result of adding the names $\{a_1, \ldots, a_n\}$ to $\mathfrak{Str}$ and binding them to the internal type names $\{b_1, \ldots, b_n\}$ already in $\mathfrak{Str}$. The result is a new strval. The names $a_1, \ldots, a_n$ are required to be atomic, so none of the substructures of $\mathfrak{Str}$ are altered.

$bind : (atid \times id)\text{-}set \times strval \rightarrow strval$

$bind(\{\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle\}, \langle tag, \xi, N, \tau, \Sigma, C\rangle) =$
  let tag' be a new tag in
  let $\tau' = \{a_1 \hookrightarrow b_1, \ldots, a_n \hookrightarrow b_n\}$ *in*
  $\langle\ tag',\ \xi,\ N \cup \{a_1, \ldots, a_n\},\ \tau \cup \tau',\ \Sigma,\ C\ \rangle$
 <u>*error*</u> *if* $N \cap \{a_1, \ldots, a_n\} \neq \phi$

6 Semantic functions

The subscripts on a few of the semantic functions indicate the context in which the functions apply (e.g. $\mathit{type}_{\mathit{Sig}}$ gives the semantics of a type identifier appearing within a signature expression **sig** ... **end**).

$\mathcal{P}rog$: *prog*
→ *strval*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*strval* × *structure-environment* × *signature-environment* × *functor-environment*)

$\mathcal{S}igb$: *sigb*
→ *signature-environment*
→ (*atid* × *sigval*)

$\mathcal{F}unb$: *funb*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*atid* × *funval*)

$\mathcal{P}list$: *plist*
→ *signature-environment*
→ (*atid-list* × *sigval*)

$\mathcal{P}atheq$: *patheq*
→ *sigval*
→ (*id* × *id*)*-set*

$\mathcal{S}trb$: *strb*
→ *strval*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*atid* × *strval*)

$\mathcal{S}ig$: *sig*
→ *signature-environment*
→ *sigval*

$\mathcal{S}pec$: *spec*
→ *sigval*
→ *signature-environment*
→ *sigval*

$\mathcal{S}pectb$: *spectb*
→ *sigval*
→ *sigval*

$\mathcal{S}pecstrb$: *specstrb*
→ *sigval*
→ *signature-environment*
→ (*atid* × *sigval*)

$\mathcal{S}tr$: *str*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ *strval*

$\mathcal{D}ec$: *dec*
→ *strval*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*strval* × *structure-environment*)

$\mathcal{D}eclist$: *declist*
→ *strval*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*strval* × *structure-environment*)

$\mathcal{T}b$: *tb*
→ *strval*
→ *structure-environment*
→ *strval*

$\mathcal{D}b_{\mathcal{S}ig}$: *db*
→ *sigval*
→ *sigval*

$\mathcal{D}b_{\mathcal{S}tr}$: *db*
→ *strval*
→ *structure-environment*
→ *strval*

$val_{\mathcal{S}ig}$: *id*
→ *sigval*
→ *internal-value-name* × *type*

$val_{\mathcal{S}tr}$: *id*
→ *strval*
→ *structure-environment*
→ *internal-value-name* × *type*

$type_{\mathcal{S}ig}$: *id*
→ *sigval*
→ *internal-type-name*

$type_{\mathcal{S}tr}$: *id*
→ *strval*
→ *structure-environment*
→ *internal-type-name*

7 Semantic equations

The result of $\mathcal{P}rog$ is a strval containing all the new (top-level) type and value bindings introduced by the program, together with any new bindings introduced into the structure, signature and functor environments. A program *prog* is interpreted in the initial environment $\langle\rho_0,\psi_0,\pi_0\rangle$ in the context of the strval *primitive-str*. The environment which results from interpreting *prog* is then the combination of *primitive-str*, ρ_0, ψ_0, π_0 and the new bindings given by $\mathscr{Prog}[\![prog]\!]\ primitive\text{-}str\ \rho_0\ \psi_0\ \pi_0$. Note that a program does not amount to an encapsulated structure declaration surrounded by an implicit **struct** ... **end**. First, signature and functor declarations are only permitted at top level, not in structure expressions. Second, the strval produced by $\mathscr{Prog}[\![dec]\!]$ does not include the type and value bindings produced by structure declarations it contains (in contrast to the strval produced by $\mathscr{Dec}[\![dec]\!]$); these contribute to the structure environment only. Finally, there is no sense in which the strval produced by $\mathscr{Prog}$ must satisfy the signature closure rule.

$$
\begin{array}{l}
\mathscr{Prog}[\![\textbf{signature}\ sigb_1\ \textbf{and}\ \ldots\ \textbf{and}\ sigb_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \\
\quad let\ \langle atid_1,\mathfrak{Sig}_1\rangle,\ldots,\langle atid_n,\mathfrak{Sig}_n\rangle = \mathscr{Sigb}[\![sigb_1]\!]\ \psi,\ \ldots,\ \mathscr{Sigb}[\![sigb_n]\!]\ \psi\ in \\
\qquad \langle\ \phi,\ \phi,\ \{atid_1 \hookrightarrow \mathfrak{Sig}_1,\ \ldots,\ atid_n \hookrightarrow \mathfrak{Sig}_n\},\ \phi\ \rangle \\
\ \ \underline{error}\ if\ atid_i{=}atid_j\ for\ some\ i{\neq}j
\end{array}
$$

$$
\begin{array}{l}
\mathscr{Prog}[\![\textbf{functor}\ funb_1\ \textbf{and}\ \ldots\ \textbf{and}\ funb_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \\
\quad let\ \langle atid_1,\mathfrak{Fun}_1\rangle,\ldots,\langle atid_n,\mathfrak{Fun}_n\rangle = \mathscr{Funb}[\![funb_1]\!]\ \rho\ \psi\ \pi,\ \ldots,\ \mathscr{Funb}[\![funb_n]\!]\ \rho\ \psi\ \pi\ in \\
\quad \langle\ \phi,\ \phi,\ \phi,\ \{atid_1 \hookrightarrow \mathfrak{Fun}_1,\ \ldots,\ atid_n \hookrightarrow \mathfrak{Fun}_n\}\ \rangle \\
\ \ \underline{error}\ if\ atid_i{=}atid_j\ for\ some\ i{\neq}j
\end{array}
$$

$$
\begin{array}{l}
\mathscr{Prog}[\![dec]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \\
\quad let\ \langle\ \mathfrak{Str}',\ \rho'\ \rangle = \mathscr{Dec}[\![dec]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi\ in \\
\quad let\ toplevelnames = \{atid \mid atid{\in}Names[\mathfrak{Str}']\}\ in \\
\quad \langle\ restrict(\mathfrak{Str}',toplevelnames),\ \rho',\ \phi,\ \phi\ \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathscr{Prog}[\![prog\ \{;\}\ prog']\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \\
\quad let\ \langle\ \mathfrak{Str}',\ \rho',\ \psi',\ \pi'\ \rangle = \mathscr{Prog}[\![prog]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi\ in \\
\quad let\ \langle\ \mathfrak{Str}'',\ \rho'',\ \psi'',\ \pi''\ \rangle = \mathscr{Prog}[\![prog']\!]\ (\mathfrak{Str} + \mathfrak{Str}')\ (\rho + \rho')\ (\psi + \psi')\ (\pi + \pi')\ in \\
\quad \langle\ \mathfrak{Str}' + \mathfrak{Str}'',\ \rho' + \rho'',\ \psi' + \psi'',\ \pi' + \pi''\ \rangle
\end{array}
$$

$$
\mathscr{Sigb}[\![atid = sig]\!]\ \psi = \langle\ atid,\ \mathscr{Sig}[\![sig]\!]\ \psi\ \rangle
$$

Functors are treated as macros in this semantics, in the sense that the body of a functor is kept in its funval as a syntactic object rather than as some sort of parameterised strval. However, the parameter declaration is processed at definition time and the functor body is checked to ensure that it is well formed and that any application will produce a valid strval with the declared signature (if one is given). The environment at declaration time must be saved for use at application time since by then some of the identifiers used in the functor body might have been bound to new values. See below under "Comments on the semantics of functors" for a semantics in which functors are "compiled" at definition time.

A functor with several parameters is treated as a functor with a single parameter having a substructure of the appropriate name for each of the several parameters. In checking whether applications of the functor will produce valid strvals which fit the declared result signature, the functor body is elaborated in a structure environment augmented by binding the formal parameter names to dummy actual parameter structures (C_{dummy} below is an arbitrary algebra of the appropriate signature, and t_{dummy} is an arbitrary tag). The result is then fitted to the declared result signature, if one is given. Note that the signature of the final result of this process differs from the declared signature in that it shares types and values with $\mathfrak{Sig}_{par}$ in a way which reflects the references which the functor body makes to the formal parameters. The signature closure rule requires the structure declared by a functor body to have no anonymous exported type names. In the form *atid(plist) : sig = str* this is guaranteed by the requirement that the body fits the given signature which must itself satisfy that rule.

$\mathcal{F}unb[\![atid(plist) = str]\!]\ \rho\ \psi\ \pi =$
$\quad let\ \langle\ atid_1\ \ldots\ atid_n,\ \mathfrak{Sig}_{par}\ \rangle = \mathcal{P}list[\![plist]\!]\ \psi\ in$
$\quad \langle\ atid,\ \langle\ atid_1 \ldots atid_n,\ \mathfrak{Sig}_{par},\ str,\ \rho,\ \psi,\ \pi\ \rangle\ \rangle$
$\quad \underline{error}\ if\ \neg signature\text{-}closed(\mathcal{S}tr[\![str]\!]\ \rho'\ \psi\ \pi)$
$\quad where\ \rho' = \rho + \{atid_1 \mapsto substructure(atid_1, \langle t_{dummy}, \mathfrak{Sig}_{par}, C_{dummy}\rangle), \ldots\}$

$\mathcal{F}unb[\![atid(plist) : sig = str]\!]\ \rho\ \psi\ \pi =$
$\quad let\ \langle\ atid_1\ \ldots\ atid_n,\ \mathfrak{Sig}_{par}\ \rangle = \mathcal{P}list[\![plist]\!]\ \psi\ in$
$\quad \langle\ atid,\ \langle\ atid_1 \ldots atid_n,\ \mathfrak{Sig}_{par},\ str{:}sig,\ \rho,\ \psi,\ \pi\ \rangle\ \rangle$
$\quad \underline{error}\ if\ \mathcal{S}tr[\![str : sig]\!]\ \rho'\ \psi\ \pi\ fails$
$\quad where\ \rho' = \rho + \{atid_1 \mapsto substructure(atid_1, \langle t_{dummy}, \mathfrak{Sig}_{par}, C_{dummy}\rangle), \ldots\}$

$\mathcal{P}list[\![atid_1 : sig_1, \ldots, atid_n : sig_n]\!]\ \psi =$
$\quad let\ \mathfrak{Sig}' = \mathcal{S}pec[\![\mathsf{structure}\ atid_1 : sig_1\ \mathsf{and}\ \ldots\ atid_n : sig_n]\!]\ Sig[primitive\text{-}str]\ \psi\ in$
$\quad \langle\ atid_1\ \ldots\ atid_n,\ Sig[primitive\text{-}str] \cup \mathfrak{Sig}'\ \rangle$

$\mathcal{P}list[\![atid_1 : sig_1, \ldots, atid_n : sig_n\ \mathsf{sharing}\ patheq_1\ \mathsf{and}\ \ldots\ patheq_m]\!]\ \psi =$
$\quad let\ \mathfrak{Sig}' = \mathcal{S}pec[\![\mathsf{structure}\ atid_1 : sig_1\ \mathsf{and}\ \ldots\ atid_n : sig_n$
$\qquad\qquad \mathsf{sharing}\ patheq_1\ \mathsf{and}\ \ldots\ patheq_m]\!]\ Sig[primitive\text{-}str]\ \psi\ in$
$\quad \langle\ atid_1\ \ldots\ atid_n,\ Sig[primitive\text{-}str] \cup \mathfrak{Sig}'\ \rangle$

$\mathcal{P}atheq[\![id_1 = \ldots = id_n]\!]\ \mathfrak{Sig} = \{\langle id_1, id_j\rangle \mid 2 \leq j \leq n\}$

A structure binding of the form *atid = str* has the effect of adding a substructure called *atid* to the environment of current bindings (by adding bindings of all the types/values in *str*, with their names prefixed by *atid*) as well as to the structure environment. The result of $\mathcal{S}trb$ is the identifier *atid* together with the strval to which it is to be bound. The environment of current bindings must not already contain a type/value with a name of the form *atid.n* since this would cause it to be regarded as a part of the new substructure. The signature closure rule requires the declared structure to have no anonymous exported type names. The form *atid : sig = str* has the same semantics, except that the structure is required to fit the declared signature. If this is the case, then the resulting structure is guaranteed to satisfy the signature closure rule.

$\mathcal{S}trb[\![atid = str]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi =$
  let $\mathfrak{Str}' = \mathcal{S}tr[\![str]\!]\ \rho\ \psi\ \pi$ in
  $\langle\ atid,\ \mathfrak{Str}'\ \rangle$
 error if $Names[\mathfrak{Str}]$ contains identifiers of the form atid.n
   or $\neg$signature-closed($\mathfrak{Str}'$)

$\mathcal{S}trb[\![atid : sig = str]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \langle\ atid,\ \mathcal{S}tr[\![str : sig]\!]\ \rho\ \psi\ \pi\ \rangle$
 error if $Names[\mathfrak{Str}]$ contains identifiers of the form atid.n

$\mathcal{S}ig[\![atid]\!]\ \psi = \psi(atid)$
 error if $atid \notin dom(\psi)$

$\mathcal{S}ig[\![$**sig** $spec$ **end**$]\!]\ \psi =$
  let $\mathfrak{Sig}' = \mathcal{S}pec[\![spec]\!]\ Sig[primitive\text{-}str]\ \psi$ in
  $Sig[primitive\text{-}str] + \mathfrak{Sig}'$

The result of $\mathcal{S}pec$ (resp. $\mathcal{S}pectb$) is a sigval containing all the new type and value bindings introduced by the specification (resp. specification type-binding), as well as the old internal type names needed to express the types of the new values. Although these will be anonymous in the immediate result of $\mathcal{S}pec$ (resp. $\mathcal{S}pectb$), names will eventually be bound to them; otherwise they would not be accessible.

$\mathcal{S}pec[\![$**val** $atid_1 : ty_1$ **and** ... **and** $atid_n : ty_n]\!]\ \mathfrak{Sig}\ \psi =$
  let newnames = $\{atid_j{:}[\![ty_j]\!] \mid [\![ty_j]\!]$ is the internal type denoted by ty_j where the
      denotation of a type name id is given by $type_{\mathcal{S}ig}[\![id]\!]\ \mathfrak{Sig}\}$ i
  let oldtypes = the (internal names of) types which newnames refers to in
  joinnames(joinanon(ϕ,oldtypes),newnames)
 error if $atid_i = atid_j$ for some $i \neq j$
   or $Names[\mathfrak{Sig}] \cap \{atid_1,\ldots,atid_n\} \neq \phi$

$\mathcal{S}pec[\![$**type** $spectb]\!]\ \mathfrak{Sig}\ \psi = \mathcal{S}pectb[\![spectb]\!]\ \mathfrak{Sig}$

$\mathcal{S}pec[\![$**datatype** $db]\!]\ \mathfrak{Sig}\ \psi = \mathcal{D}b_{\mathcal{S}ig}[\![db]\!]\ \mathfrak{Sig}$

$\mathcal{S}pec[\![spec\ \{;\}\ spec']\!]\ \mathfrak{Sig}\ \psi =$
  let $\mathfrak{Sig}' = \mathcal{S}pec[\![spec]\!]\ \mathfrak{Sig}\ \psi$ in
  let $\mathfrak{Sig}'' = \mathcal{S}pec[\![spec']\!]\ (\mathfrak{Sig} + \mathfrak{Sig}')\ \psi$ in
  $\mathfrak{Sig}' + \mathfrak{Sig}''$

Types and values declared in a structure binding contribute to the environment of current bindings, with names prefixed by the name of the (sub)structure in which they appear.

$\mathcal{S}pec[\![$**structure** $specstrb_1$ **and** ... **and** $specstrb_n]\!]\ \mathfrak{Sig}\ \psi =$
  let $\langle atid_1,\mathfrak{Sig}_1\rangle,\ldots,\langle atid_n,\mathfrak{Sig}_n\rangle = \mathcal{S}pecstrb[\![specstrb_1]\!]\ \mathfrak{Sig}\ \psi,\ \ldots,\ \mathcal{S}pecstrb[\![specstrb_n]\!]\ \mathfrak{Sig}\ \psi$ in
  addsubstrs($\{\langle atid_1,\mathfrak{Sig}_1\rangle,\ldots,\langle atid_n,\mathfrak{Sig}_n\rangle\},\phi$)
 error if $atid_i = atid_j$ for some $i \neq j$

$\mathcal{Spec}[\![$**structure** $specstrb_1$ **and** ... **and** $specstrb_n$ **sharing** $patheq_1$ **and** ... **and** $patheq_m]\!]$ $\mathfrak{Sig}\ \psi =$
  $let\ \langle atid_1,\mathfrak{Sig}_1\rangle,...,\langle atid_n,\mathfrak{Sig}_n\rangle = \mathcal{Specstrb}[\![specstrb_1]\!]\ \mathfrak{Sig}\ \psi,\ ...,\ \mathcal{Specstrb}[\![specstrb_n]\!]\ \mathfrak{Sig}\ \psi\ in$
  $let\ \mathfrak{Sig}' = addsubstrs(\{\langle atid_1,\mathfrak{Sig}_1\rangle,...,\langle atid_n,\mathfrak{Sig}_n\rangle\},\phi)\ in$
  $identify(\ \mathcal{Patheq}[\![patheq_1]\!]\ \mathfrak{Sig}' \cup ... \cup \mathcal{Patheq}[\![patheq_m]\!]\ \mathfrak{Sig}',\ \mathfrak{Sig}'\ )$
 <u>*error*</u> $if\ atid_i{=}atid_j\ for\ some\ i{\neq}j$

$\mathcal{Spectb}[\![atid]\!]\ \mathfrak{Sig} = joinnames(\phi,\{atid\})$
 <u>*error*</u> $if\ atid{\in}Names[\mathfrak{Sig}]$

$\mathcal{Spectb}[\![atid = id]\!]\ \mathfrak{Sig} =$
  $let\ idmeaning = \mathcal{type}_{\mathcal{Sig}}[\![id]\!]\ \mathfrak{Sig}\ in$
  $bind(\{\langle atid,idmeaning\rangle\},joinanon(\phi,\{idmeaning\}))$
 <u>*error*</u> $if\ atid{\in}Names[\mathfrak{Sig}]$

$\mathcal{Spectb}[\![spectb$ **and** $spectb']\!]\ \mathfrak{Sig} =$
  $let\ \mathfrak{Sig}' = \mathcal{Spectb}[\![spectb]\!]\ \mathfrak{Sig}\ in$
  $let\ \mathfrak{Sig}'' = \mathcal{Spectb}[\![spectb']\!]\ \mathfrak{Sig}\ in$
  $\mathfrak{Sig}' + \mathfrak{Sig}''$
 <u>*error*</u> $if\ Names[\mathfrak{Sig}'] \cap Names[\mathfrak{Sig}''] \neq \phi$

Structure bindings of the form *atid* **:** *sig* are permitted in specification contexts only. A substructure called *atid* containing the types and values in *sig* contributes to the environment of current type and value bindings. These types and values are forced to be distinct from all the types and values already present, with the exception of pervasive types/values. That this is necessary is shown by the example declaration **structure A :** *sig* **and B :** *sig* , since **A.**n is not expected to share with **B.**n for a type/value n in *sig* (unless e.g. n=**bool**).

$\mathcal{Specstrb}[\![atid\ :\ sig]\!]\ \mathfrak{Sig}\ \psi = \langle\ atid,\ tag(\mathcal{Sig}[\![sig]\!]\ \psi,AlgSig[primitive\text{-}str])\ \rangle$
 <u>*error*</u> $if\ Names[\mathfrak{Sig}]\ contains\ identifiers\ of\ the\ form\ atid.n$

According to the closure rule for structures, a structure expression is allowed to contain references to previously-defined structures, signatures and functors but not to current type/value bindings (except to pervasive primitives from *primitive-str*); consequently $\mathcal{Str}$ does not require access to the current strval. The result of $\mathcal{Str}$ does not necessarily satisfy the signature closure rule (which implies that structures have no anonymous exported type names) since the result of fitting a non-complying structure to a signature will satisfy that rule.

$\mathcal{Str}[\![atid]\!]\ \rho\ \psi\ \pi = \rho(atid)$
 <u>*error*</u> $if\ atid{\notin}dom(\rho)$

$\mathcal{Str}[\![atid.id]\!]\ \rho\ \psi\ \pi = substructure(id,\rho(atid))$
 <u>*error*</u> $if\ atid{\notin}dom(\rho)$

$\mathcal{S}tr$[[**struct** *declist* **end**]] $\rho\ \psi\ \pi$ =
  let ⟨ 𝔖𝔱𝔯, ρ' ⟩ = $\mathcal{D}eclist$[[*declist*]] *primitive-str* $\rho\ \psi\ \pi$ *in*
  primitive-str + 𝔖𝔱𝔯

$\mathcal{S}tr$[[*str* : *sig*]] $\rho\ \psi\ \pi$ = *fit*($\mathcal{S}tr$[[*str*]] $\rho\ \psi\ \pi$,$\mathcal{S}ig$[[*sig*]] ψ)

The result of applying a functor to a list of actual parameters is obtained by elaborating the body of the functor in the declaration-time environment augmented by binding the parameter names to the actual parameters (after fitting them to the formal parameter signatures).

$\mathcal{S}tr$[[*atid*(str_1, ..., str_n)]] $\rho\ \psi\ \pi$ =
  let ⟨ $atid_1 \ldots atid_m$, $\mathfrak{Sig}_{par}$, *str*, ρ', ψ', π' ⟩ = π(*atid*) *in*
  let 𝔖𝔱𝔯' = *fit*(*addsubstrs*({⟨$atid_1$,$\mathcal{S}tr$[[str_1]] $\rho\ \psi\ \pi$⟩,
    . . . ,
    ⟨$atid_n$,$\mathcal{S}tr$[[str_n]] $\rho\ \psi\ \pi$⟩},*primitive-str*), $\mathfrak{Sig}_{par}$) *in*
  $\mathcal{S}tr$[[*str*]] (ρ' + {$atid_1$⟼*substructure*($atid_1$,𝔖𝔱𝔯'),...}) $\psi'\ \pi'$
  <u>*error*</u> *if* $atid \notin dom(\pi)$
    or if $n \neq m$

The semantics of type and value declarations is incomplete in that it does not give all the details of the interpretation of the type/value bindings themselves, but only the details of how they contribute to the environment of current bindings. The result of $\mathcal{D}ec$ is a strval containing all the new bindings introduced by the declaration as well as the old internal type names needed to express the types of the new values, together with any new bindings introduced into the structure environment. It is possible to cause anonymous types to be added to the current structure with a declaration like `val f=A.g` or `datatype t = f of A.t1` . Names must eventually be bound to these types or else the signature closure rule will be broken.

$\mathcal{D}ec$[[**val** *vb*]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ =
  let newnames = the identifiers which vb binds, with their types (using internal type n
  let oldtypes = the (internal names of) types which newnames refers to in
  let 𝔖𝔦𝔤 = *joinnames*(*joinanon*(ϕ,*oldtypes*),*newnames*) *in*
  let C = the AlgSig[𝔖𝔦𝔤]*-algebra defined by vb (including carriers associated with*
    oldtypes, taken from 𝔖𝔱𝔯) *in*
  ⟨ ⟨𝔖𝔦𝔤,*C*⟩, ϕ ⟩

$\mathcal{D}ec$[[**type** *tb*]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ = ⟨ $\mathcal{T}b$[[*tb*]] 𝔖𝔱𝔯 ρ, ϕ ⟩

$\mathcal{D}ec$[[**datatype** *db*]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ = ⟨ $\mathcal{D}b_{\mathcal{S}tr}$[[*db*]] 𝔖𝔱𝔯 ρ, ϕ ⟩

$\mathcal{D}ec$[[**local** *declist* **in** *declist'* **end**]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ =
  let ⟨ 𝔖𝔱𝔯', ρ' ⟩ = $\mathcal{D}eclist$[[*declist*]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ *in*
  $\mathcal{D}eclist$[[*declist'*]] (𝔖𝔱𝔯 + 𝔖𝔱𝔯') (ρ + ρ') $\psi\ \pi$

$\mathcal{D}ec$[[**open** *id*]] 𝔖𝔱𝔯 $\rho\ \psi\ \pi$ = ⟨ *substructure*(*id*,𝔖𝔱𝔯), ϕ ⟩

$\mathcal{D}ec[\![$open $id_1\ id_2\ \ldots\ id_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \mathcal{D}eclist[\![$open id_1; open id_2; ...; open $id_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi$

Types and values declared in a structure binding contribute to the environment of current bindings. The newly-declared structure also contributes to the structure environment for the benefit of nested encapsulated structure declarations, to which it appears as a previously-defined structure. Sharing constraints are not permitted in structure contexts; sharing in a structure arises by construction rather than by declaration.

$\mathcal{D}ec[\![$structure $strb_1$ and ... and $strb_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi =$
  let $\langle atid_1,\mathfrak{Str}_1\rangle,\ldots,\langle atid_n,\mathfrak{Str}_n\rangle = \mathcal{S}trb[\![strb_1]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi,\ldots,\mathcal{S}trb[\![strb_n]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi$ *in*
  $\langle\ addsubstrs(\{\langle atid_1,\mathfrak{Str}_1\rangle,\ldots,\langle atid_n,\mathfrak{Str}_n\rangle\},\phi),\ \{atid_1 \hookrightarrow \mathfrak{Str}_1,\ldots,atid_n \hookrightarrow \mathfrak{Str}_n\}\ \rangle$
 error if $atid_i = atid_j$ *for some* $i \neq j$

$\mathcal{D}eclist[\![dec]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi = \mathcal{D}ec[\![dec]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi$

$\mathcal{D}eclist[\![declist\ \{;\}\ declist']\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi =$
  let $\langle\ \mathfrak{Str}',\ \rho'\ \rangle = \mathcal{D}eclist[\![declist]\!]\ \mathfrak{Str}\ \rho\ \psi\ \pi$ *in*
  let $\langle\ \mathfrak{Str}'',\ \rho''\ \rangle = \mathcal{D}eclist[\![declist']\!]\ (\mathfrak{Str} + \mathfrak{Str}')\ (\rho + \rho')\ \psi\ \pi$ *in*
  $\langle\ \mathfrak{Str}' + \mathfrak{Str}'',\ \rho' + \rho''\ \rangle$

The interpretation of type, datatype and value bindings is as in the core language, subject to the provision that the denotation of a reference to a previously-defined type/value is given by the semantic functions $type_{\mathcal{S}tr}/val_{\mathcal{S}tr}$. This semantics does not keep track of which values are constructors, but this would be an easy refinement to add.

$\mathcal{T}b[\![atid = id]\!]\ \mathfrak{Str}\ \rho =$
  let $idmeaning = type_{\mathcal{S}tr}[\![id]\!]\ \mathfrak{Str}\ \rho$ *in*
  let $\mathfrak{Sig} = bind(\{\langle atid,idmeaning\rangle\},joinanon(\phi,\{idmeaning\}))$ *in*
  let C = *the* $AlgSig[\mathfrak{Sig}]$-*algebra with* $|C|_{idmeaning} = |Alg[\mathfrak{Str}]|_{idmeaning}$ *in*
  $\langle\mathfrak{Sig},C\rangle$

$\mathcal{T}b[\![tb$ and $tb']\!]\ \mathfrak{Str}\ \rho = (\mathcal{T}b[\![tb]\!]\ \mathfrak{Str}\ \rho) + (\mathcal{T}b[\![tb']\!]\ \mathfrak{Str}\ \rho)$

$\mathcal{D}b_{\mathcal{S}ig}[\![atid_1 = constrs_1$ and ... and $atid_n = constrs_n]\!]\ \mathfrak{Sig} =$[7]
  let $\mathfrak{Sig}' = joinnames(\mathfrak{Sig},\{atid_1,\ldots,atid_n\})$ *in*
  let $newvals$ = *the constructors in* $constrs_1,\ldots,constrs_n$ *together with their types in*
  let $oldtypes$ = *the internal names of types which newvals refers to in*
  $joinnames(joinanon(\mathfrak{Sig}',oldtypes),newvals)$
 error if $atid_i = atid_j$ *for some* $i \neq j$
   or $Names[constrs_i] \cap Names[constrs_j] \neq \phi$ *for some* $i \neq j$
   or $\{atid_1,\ldots,atid_n\} \cap Names[newvals] \neq \phi$
   or $Names[\mathfrak{Sig}] \cap (\{atid_1,\ldots,atid_n\} \cup Names[newvals]) \neq \phi$

---

[7] $\mathcal{D}b_{\mathcal{S}ig}$ and $\mathcal{D}b_{\mathcal{S}tr}$ should be changed to return only the *new* types and values; at present they return the old types/values as well.

$\mathcal{D}b_{Str}[\![atid_1 = constrs_1$ **and** ... **and** $atid_n = constrs_n]\!]$ $\mathfrak{Str}$ ρ =
  let $\mathfrak{Sig}' = joinnames(Sig[\mathfrak{Str}],\{atid_1,...,atid_n\})$ *in*
  let $newvals$ = *the constructors in* $constrs_1,...,constrs_n$ *together with their types in*
  let $oldtypes$ = *the internal names of types which newvals refers to in*
  let $\mathfrak{Sig}'' = joinnames(joinanon(\mathfrak{Sig}',oldtypes),newvals)$ *in*
  let C'' = *the* $AlgSig[\mathfrak{Sig}'']$*-algebra defined by the type and constructor bindings*
    (including carriers associated with oldtypes, taken from $\mathfrak{Str}$*) in*
  $\langle\mathfrak{Sig}'',C''\rangle$

The functions val_{Sig}, $type_{Sig}$ and val_{Str}, $type_{Str}$ interpret value and type names in signature and structure contexts respectively. The difference between these contexts is a consequence of the different closure rules for signatures and structures; while structures may contain references to elements of previously-defined structures, signatures may not contain such references. These functions return the internal name of the value or type referenced.

$val_{Sig}[\![id]\!]\langle\xi,N,\tau,\Sigma\rangle = \tau(id)$
  <u>error</u> *if* $id \notin N$
    or $\tau(id) \notin Vals[\Sigma]$

$val_{Str}[\![atid]\!]\langle tag,\xi,N,\tau,\Sigma,C\rangle = \tau(atid)$
  <u>error</u> *if* $atid \notin N$
    or $\tau(atid) \notin Vals[\Sigma]$

$val_{Str}[\![atid.id]\!]$ $\mathfrak{Str}$ ρ =
  let $\langle tag',\xi',N',\tau',\Sigma',C'\rangle = \rho(atid)$ *in* $\tau'(id)$
  <u>error</u> *if* $atid \notin dom(\rho)$
    or $id \notin N'$
    or $\tau'(id) \notin Vals[\Sigma']$

$type_{Sig}[\![id]\!]\langle\xi,N,\tau,\Sigma\rangle = \tau(id)$
  <u>error</u> *if* $id \notin N$
    or $\tau(id) \notin Types[\Sigma]$

$type_{Str}[\![atid]\!]\langle tag,\xi,N,\tau,\Sigma,C\rangle = \tau(atid)$
  <u>error</u> *if* $atid \notin N$
    or $\tau(atid) \notin Types[\Sigma]$

$type_{Str}[\![atid.id]\!]$ $\mathfrak{Str}$ ρ =
  let $\langle tag',\xi',N',\tau',\Sigma',C'\rangle = \rho(atid)$ *in* $\tau'(id)$
  <u>error</u> *if* $atid \notin dom(\rho)$
    or $id \notin N'$
    or $\tau'(id) \notin Types[\Sigma']$