A Semantics for Higher-order Functors

David B. MacQueen[1] and Mads Tofte[2]

[1] AT&T Bell Labs, New Jersey, USA
[2] Dept. of Computer Science, Copenhagen University, Denmark

Abstract. Standard ML has a module system that allows one to define parametric modules, called *functors*. Functors are "first-order," meaning that functors themselves cannot be passed as parameters or returned as results of functor applications. This paper presents a semantics for a higher-order module system which generalizes the module system of Standard ML. The higher-order functors described here are implemented in the current version of Standard ML of New Jersey and have proved useful in programming practice.

1 Introduction

One of the notable characteristics of the Standard ML module system has been its support of parameterization in the form of *functors*, which are mappings from ordinary modules, called *structures*, to ordinary modules. In the original Standard ML module system ([7, 10]), functors were first-order, because their parameters and results could only be structures, and functors could not be components of structures. But the type theoretic analysis of the module system carried out in [8, 11, 5] made it clear that it was natural to extend the notion of functors to higher orders by allowing functors as parameters and results (or, equivalently, allowing structures to contain functor components). Doing so makes the language more symmetrical and supports useful new modes of parameterization.

A practical implementation of higher-order functors has recently been provided in the Standard ML of New Jersey compiler [3]. The first step toward defining a semantics of higher-order functors was taken in [14], where a semantics for functor signatures is described and a principal signature theorem is proved. Here we go most of the way toward completing the semantics of higher-order functors by defining how functors are represented, how higher-order signature matching is performed, and how functor application works.

The technical challenge in defining a semantics of higher-order functors arises from the way static identity information is propagated in Standard ML. Signature matching is "transparent" by default, meaning that the identities of type and structure components are not hidden when a structure is matched against a signature. Also, identities are propagated through functor calls. This is a controversial feature of the design, but it is justified because (1) it allows a single semantics of signature matching to work both for parameter constraints and result constraints, and (2) it increases the expressiveness and flexibility of parameterization in useful ways.

Alternative module system designs that do not use transparent signature matching have been proposed. For instance, the Extended ML specification language [13], which is based on Standard ML, assumes that signature matching is opaque, and recently Leroy [6] and Harper and Lillibridge [4] have described module systems that use opaque signature matching but allow one to override it in the case of types by using type definitions in signatures. However, in the higher-order system these proposals produce an asymmetry between first-order and higher-order functors: types in a functor result can depend on structure parameters, but not on functor parameters (see the example below).

We want to avoid this asymmetry between structures and functors, so in our semantics functor parameters as well as structure parameters can carry type information that is propagated to the result.

As an illustration of how application of a first-order functor propagates static identities, consider the following example:

```
signature POINT =
  sig
    type point
    val leq: point*point->bool
  end;
signature INTERVAL =
  sig
    type interval and point
    val mk: point*point -> interval
    val left: interval -> point
    val right: interval -> point
  end;
functor Interval(P: POINT): INTERVAL =
  struct
    type interval = P.point * P.point
    type point = P.point
    fun mk(x,y) = if P.leq(x,y) then (x,y) else (y,x)
    fun left(x,_) = x
    fun right(_,y) = y
  end;
structure IntPoint =
  struct
    type point = int; (*1*)
    fun leq(x:int,y) = x<=y
  end;
structure T = Interval(IntPoint);
val test = T.right(T.mk(3,4))+5;
```

This program is legal Standard ML. The declaration of `test` is type-correct because the application `Interval(IntPoint)` propagates the information `point = int` (declared at line `(*1*)`) through to `T`, so that `T.interval` is `int*int` and `T.mk` and `T.right` have types `int*int->int*int` and `int*int->int`, respectively. (Notice that if types were *not* propagated through the functor application, the declaration of `test` would be illegal.)

Now let us add a higher-order functor:

```
functor G(functor Interv(P: POINT): INTERVAL) =
  struct
    structure NatNumInt = Interv(IntPoint)
  end
```

Since the actual functor Interval matches the specification in the parameter signature for G, it should be possible to apply G to Interval:

```
structure Result = G(functor Interv = Interval)
structure  T' =  Result.NatNumInt;
val test' = T'.right(T'.mk(3,4))+5
```

But will the expression `T'.right(T'.mk(3,4))+5` be type-correct? The point is that the parameter signature of G did not specify sharing between the argument and the result signature of Interv. Thus when the declaration of G was elaborated, there was no assumption of sharing between the point type P.point and the point type NatNumInt.point.

The actual functor, Interval, propagates more sharing than is specified for Interv. Were we to elaborate the body of G again, this time using the actual Interval in place of Interv, the declaration of test' would be legal; if we ignore the extra sharing, however, the declaration of test' becomes untypable.

One could argue that this problem is easily solved by making the specification of Interv more specific so that it expresses the sharing required:

```
functor G(X:
  sig
    functor Interv(P: POINT):
      sig
        type interval
        val mk: P.point * P.point -> interval
        val left: interval -> P.point
        val right: interval -> P.point
      end
  end)=
struct
  structure NatNumInt = Interv(IntPoint)
end
```

But after this change we can only apply G to arguments that satisfy this extra sharing, which was not needed inside the body of the functor G, so G is less general than it could be.

More generally, consider the declaration of some functor, F. Is it sufficient to specify the parameter signature of F with sharing constraints that are needed to elaborate the body of F, or is it necessary to specify any sharing that must be propagated at some application of F? From a programmer's point of view, the former is clearly preferable and it is the policy followed in Standard ML. To preserve this desirable property of Standard ML in the presence of higher-order functors, our static semantics of modules must be able to propagate additional type information at functor application time, even when the additional information comes from functor components of the actual argument. So to properly treat a functor application embedded in a functor body, such as Interv(IntPoint) in the body of G, we must elaborate it in two phases: first formally, when G is defined, and then again when G is applied and additional sharing information about the actual parameter is available.

In the remainder of this paper we first present the semantic objects (Section 2). Then we give a grammar for a skeletal programming language and elaboration rules in terms of the semantic objects (Section 3). The key ideas for achieving the desired propagation of type information are (1) using terms in a simple higher-order language to represent functors, and (2) using two environments to simulate the two phase elaboration of embedded functor applications.

The skeletal language we present has neither types nor values, but we foresee no serious problems in extending the semantics to cope with these because the interaction between module systems and the core ML language is well understood. We also do not deal with elaboration of signature expressions in this paper; this was studied in detail in [14].

In addition to the work on Extended ML and the work of Leroy and Harper and Lillibridge already cited, the work of Aponte [1] should also be noted. It provides another approach to semantic representations for ML modules, based on Rémy's work on polymorphic records [12]. So far, this approach deals with first-order functors only.

2 Semantic objects

Our semantic objects are defined informally using a mixture of simple mathematical constructions (*e.g.* sets of sequences of identifiers) and term structures (*e.g.* lambda abstractions) over these constructions. The representations are finite, and in principle they could all be defined uniformly by an abstract syntax of terms.

In the skeletal language, a structure S can have two kinds of named components: structures and functors. The *substructures* of S are S and the substructures of the structure components of S. We say that a functor is *(embedded) in* a structure S, if it is a component of S or of one of the substructures of S. Our representation of structures is based on separating the "shape" of a structure, which defines what is accessible, from the static information that identifies the elements of the structure. The former is represented by a tree s (Section 2.1) of access paths for substructures and embedded functors, and the latter by a realization φ (Section 2.3), which represents a mapping from these paths to identifying information. A *structure* is then defined to be a pair $\langle s, \varphi\rangle$.

2.1 Identifiers, paths, and trees

We assume two disjoint sets of identifiers:

funid	$\in$	FunId	(functor identifier)
strid	$\in$	StrId	(structure identifier)

Substructures and embedded functors are accessed via paths of identifiers. Formally, a *structure path*, *sp*, is a finite string over the alphabet StrId. We also use the notation $strid_1.\cdots.strid_k$, $(k \geq 0)$ for structure paths. A *functor path*, *fp*, is a finite string over the alphabet StrId $\cup$ FunId of the form $strid_1.\cdots.strid_k.funid$, $(k \geq 0)$, *i.e.*, a structure path followed by a functor identifier. A *path*, *p*, is either a structure path or a functor path. The empty path is denoted ϵ.

A *tree*, s, is a finite, prefix-closed set of paths. Let s be a tree and assume $p \in s$. Then the *subtree of s at p*, written s/p, is the tree $\{p' \mid pp' \in s\}$. When s is a tree, $SP(s)$ denotes the set of structure paths in s and $FP(s)$ denotes the set of functor paths in s.

2.2 Stamps

Stamps are used to statically identify structures, and are the basis for determining sharing: two structures share if and only if they are labeled by the same stamp. Only structures have stamps — functors do not have a static identity, though they do have static descriptions.

We assume a denumerably infinite set Stamp of *stamps*. We use m to range over stamps. A *stamp set* is a finite set of stamps. We use M and N to range over stamp sets.

2.3 Realizations

Intuitively, the realization part of a structure is a mapping over the structure's path tree that takes structure paths to stamps and functor paths to static functor representations. However, it turns out to be useful to talk about realization expressions, rather than the maps they denote; realization expressions are defined in Figure 1. Signatures (Σ) will be defined in Section 2.4.

Realization environments and views are concrete representations of finite maps. The *domain* of a realization environment β, written $\mathrm{Dom}(\beta)$, is defined by: $\mathrm{Dom}(\{\}) = \emptyset$ and $\mathrm{Dom}(\beta', \rho{=}\varphi) = \{\rho\} \cup \mathrm{Dom}(\beta')$ and similarly for views. We allow repeated binding of the same domain element, with the convention that bindings to the right supersede bindings to the left. We write, for example, $\beta(\rho)$ to denote the realization to which β binds ρ, when $\rho \in \mathrm{Dom}(\beta)$. We often write realization environments out in full, with the notation $\rho_1{=}\varphi_1, \ldots, \rho_n{=}\varphi_n$ (dropping the initial $\{\}$). Realization environments β_1 and β_2 can be appended, written $\beta_1 + \beta_2$.

Realization Expressions (φ)

φ	$::=$	(μ, η)	stamping
	$\mid$	ρ	realization variable
	$\mid$	φ/strid	substructure selection
	$\mid$	$\mathrm{app}(\theta, \varphi)$	functor application
	$\mid$	$\varphi \downarrow \Sigma$	signature constraint
	$\mid$	$\texttt{new}\ N\,.\,\varphi$	generative stamps
	$\mid$	$\texttt{let}\ \beta\ \texttt{in}\ \varphi$	local binding

Realization Environments (β)

β	$::=$	$\{\}$	empty environment
	$\mid$	$\beta, \rho{=}\varphi$	extension

Views (η)

η	$::=$	$\{\}$	empty
	$\mid$	$\eta, \mathit{strid}{=}\varphi$	extension
	$\mid$	$\eta, \mathit{funid}{=}\theta$	extension

Static Functors (θ)

θ	$::=$	$\lambda\rho : \Sigma.\langle s, \varphi\rangle$	functor
	$\mid$	$\mathtt{get}_F(\varphi, \mathit{fp})$	application

Stamp Expressions (μ)

μ	$::=$	m	stamp
	$\mid$	$\mathtt{get}_S(\varphi, \mathit{sp})$	application

Fig. 1. Basic semantic representations

To get an idea of the meaning of the constructs of Figure 1, consider the following functor declaration:

```
functor G(X: sig
               functor Interv(P: POINT): INTERVAL
               structure I: POINT
             end)=
struct
  structure NatNum = X.Interv(X.I)
end;
```

This declaration gives rise to the view $\eta_G \equiv \mathtt{G}{=}\lambda\rho : \Sigma_X.\langle s_{\text{body}}, \varphi_{\text{body}}\rangle$, where s_{body} is the tree $\{\epsilon, \mathtt{NatNum}\}$ and

$$\varphi_{\text{body}} = \mathtt{new}\{m\}.\mathtt{let}\ \rho' = \mathtt{app}(\mathtt{get}_F(\rho, \mathtt{Interv}), (\rho/\mathtt{I}))\\ \qquad\qquad \mathtt{in}\ (m, \mathtt{NatNum}{=}\rho')$$

and Σ_X represents the parameter signature of G. Here $\mathtt{app}(\mathtt{get}_F(\rho, \mathtt{Interv}), \rho/\mathtt{I})$ stands for a functor application. The operator, $\mathtt{get}_F(\rho, \mathtt{Interv})$, is the functor component named Interv of the (unknown) realization ρ; the operand, $\rho/\mathtt{I}$, is the realization of the substructure named I.

Functor applications can generate fresh structures. For example, every application of the functor G gives rise to one fresh structure, *i.e.*. to one structure with a fresh stamp, corresponding to the expression struct $\cdots$ end forming the body of the functor. The realization expression $\mathtt{new}\ N\,.\,\varphi$ is used for expressing generativity. The stamps in N are bound in φ, and the semantic rules will force alpha-conversion to insure that these are replaced by "fresh" stamps when the functor is applied.

2.4 Signatures

Module interfaces are called signatures in Standard ML. A key feature is the ability to specify sharing in signatures. This is particularly important in connection with functors, as a means of stipulating sharing within the formal parameter structure. There are two forms of sharing in Standard ML: *structure sharing* and *type sharing*. The present semantics deals with structure sharing (but not with type sharing, as this requires integration with the Core language semantics.) Since functors do not have static identities, there is no notion of functor sharing specifications. As in Standard ML, a specification that two structures share is implicitly a specification that all substructures visible in both structures share as well. However, this does not imply that common functor components have the same functor signature. No attempt is made to "unify" functor signatures; indeed, there are valid signatures which cannot be matched by any real structure, because the signature imposes conflicting signatures on a specified functor. In this respect, functor specifications resemble the value specifications of Standard ML.

A functor specification can contain sharing specifications that impose sharing between the argument structure and the result structure, or between either of these and some structure declared or specified elsewhere. In that sense, sharing specifications can constrain a functor. A more detailed study of sharing, including functor sharing, is given in [14].

Formally, we represent sharing specifications by relations on structure paths, as follows. Let s be a tree. A *sharing relation (on s)* is a relation R satisfying:

1. R is an equivalence relation on $SP(s)$;

2. R is closed under structure path extension: for all sp, sp', $strid$, if $sp\ R\ sp'$ and $sp.strid \in s$ and $sp'.strid \in s$ then $sp.strid\ R\ sp'.strid$;
3. there are no cycles in the graph obtained by collapsing R-equivalent paths into a single node.

The equivalence class containing sp is written $[sp]_R$, or just $[sp]$, when R is clear from the context. The set of equivalence classes is denoted s/R. Given any relation R on structure paths, $Cl(R)$ is the equivalence "closure" of this relation, *i.e.* the smallest equivalence relation on $SP(s)$ containing R.

A *signature* Σ is a tuple $\langle s, R, \sigma, \delta\rho.\Phi\rangle$. Here s is a tree, R is a sharing relation on s, σ is an (external) sharing map with $\mathrm{Dom}(\sigma) \subseteq SP(s)$ mapping structure paths to stamp expressions, and Φ is a functor signature environment with $\mathrm{Dom}(\Phi) = FP(s)$ mapping functor paths to functor signatures. The δ is a binding operator binding ρ with scope Φ, and the idea is that ρ represents the realization of a hypothetical structure matching the entire signature. It is used to express sharing between an embedded functor whose signature, Ξ, is given by Φ and a substructure specified elsewhere in the signature. This sharing is represented by a free occurrence of the stamp expression $\rho(sp)$ within Ξ. Accordingly, we require that the only free occurrences of ρ in Φ are in stamp expressions of the form $\rho(sp)$, where $sp \in s$.

The following example illustrates the roles of R and σ in representing internal and external sharing in signatures.

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

The representation of this signature is $\Sigma = \langle s, R, \sigma, \delta\rho.\{\}\rangle$, where $s = \{\epsilon, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$, $R = Cl(\{(\mathtt{B}, \mathtt{C})\})$ and $\sigma = \{\mathtt{A} \mapsto m\}$, where m is the stamp of S.

We require that σ be consistent with R, so that it can be regarded as a partial map from s/R to stamp expressions, *i.e.* that if $sp\ R\ sp'$ then $\sigma(sp) = \sigma(sp')$. Furthermore, we require that the domain of σ is closed under path extension: if $sp \in \mathrm{Dom}(\sigma)$ and $sp.strid \in s$ then $sp.strid \in \mathrm{Dom}(\sigma)$.

2.5 Functor signatures

A *functor signature* Ξ takes the form $\lambda\rho : \Sigma.\Sigma_r$. Here Σ is the argument signature and Σ_r is the result signature. Write Σ_r in the form $\langle s_r, R_r, \sigma_r, \delta\rho_r.\Phi_r\rangle$. Sharing between argument and result is expressed by occurrences of stamp expressions of the form $\mathtt{get}_S(\rho, sp)$ in σ_r and Φ_r, for some sp. We require that the only free occurrences of ρ in σ_r and Φ_r are in stamp expressions of the form $\mathtt{get}_S(\rho, sp)$, where sp has to be a member of the tree component of Σ. This is to ensure proper propagation of sharing at functor application time.

Here is a functor specification illustrating propagation of information from the parameter of a functor to the result via the λ-bound realization variable in the functor signature.

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

The corresponding functor signature is $\Xi = \lambda\rho : \Sigma_X.\Sigma_R$, where $\Sigma_X = \langle\{\epsilon, \mathtt{A}\}, I, \{\}, \delta\rho'.\{\}\rangle$ and $\Sigma_R = \langle\{\epsilon, \mathtt{B}\}, I, \{\mathtt{B} \mapsto \mathtt{get}_S(\rho, \mathtt{A})\}, \delta\rho''.\{\}\rangle$ and I is the identity relation.

The next example illustrates the use of the δ-bound realization variable in expressing sharing between part of a functor and a structure specified in the same signature:

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

for which the representation is $\Sigma = \langle s, I, \sigma, \delta\rho.\{\mathtt{F} \mapsto \Xi\}\rangle$, where $s = \{\epsilon, \mathtt{A}, \mathtt{F}\}$, σ is the empty map, and $\Xi = \lambda\rho' : \Sigma_X.\Sigma_R$, with $\Sigma_X = \langle\{\epsilon, \mathtt{B}\}, I, \{[\mathtt{B}] \mapsto \mathtt{get}_S(\rho, \mathtt{A})\}, \delta\rho_1.\{\}\rangle$ and $\Sigma_R = \langle\{\epsilon\}, I, \{\}, \delta\rho_2.\{\}\rangle$.

The requirement that a δ-bound ρ only be "applied" to valid paths of the containing signature is significant for getting a well-defined notion of structure matching. Unfortunately, it also means that there is not a perfect correspondence between the present signatures and the so-called principal signatures inferred in [14]. In the latter case, one is allowed to write for example

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

in which `A'` is specified to share with `A`, although there is no specification of a `B` component of `A` outside the specification of `F`. Because of the requirement we are discussing, the principal signature for the above signature expression cannot be represented as a signature in the present semantics. Principal signatures that do not have such dangling components *can* be represented, however. Since these dangling components are easy to detect in principal signatures and could be banned without any dramatic loss in programming convenience, the two forms of signatures are not in serious conflict.

2.6 Evaluation of stamp expressions

Since we verify sharing specifications by comparing stamps, it is necessary to "evaluate" arbitrary stamp expressions to reduce them to concrete stamps. Since stamp expressions may contain realization variables, this evaluation must be performed in the context of a realization environment that binds these variables. Here are the rules defining evaluation of stamp expressions:

Stamp expressions $\boxed{\beta \vdash \mu \Rightarrow m}$

$$\frac{}{\beta \vdash m \Rightarrow m} \tag{1}$$

$$\frac{\Sigma = \langle s, R, \sigma, \delta\rho.\Phi\rangle \quad sp \in s \quad \beta \vdash \mathtt{get}_S(\varphi, sp) \Rightarrow m}{\beta \vdash (\mathtt{get}_S(\varphi \downarrow \Sigma), sp) \Rightarrow m} \tag{2}$$

$$\frac{\beta \vdash \mathtt{get}_S(\varphi, strid.sp) \Rightarrow m}{\beta \vdash \mathtt{get}_S(\varphi/strid, sp) \Rightarrow m} \tag{3}$$

$$\frac{\beta(\rho) = \varphi \quad \beta \vdash \mathtt{get}_S(\varphi, sp) \Rightarrow m}{\beta \vdash \mathtt{get}_S(\rho, sp) \Rightarrow m} \tag{4}$$

$$\frac{\varphi = (\mu, \_) \quad \beta \vdash \mu \Rightarrow m}{\beta \vdash \mathtt{get}_S(\varphi, \epsilon) \Rightarrow m} \tag{5}$$

$$\frac{\varphi = (\mu, \eta) \quad \eta(strid) = \varphi' \quad \beta \vdash \mathtt{get}_S(\varphi', sp) \Rightarrow m}{\beta \vdash \mathtt{get}_S(\varphi, strid.sp) \Rightarrow m} \tag{6}$$

We define a function $\mathit{Eval} : \mathrm{RealizationEnv} \to (\mathrm{StampExp} \rightharpoonup \mathrm{Stamp})$ by

$$\mathit{Eval}(\beta)(\mu) = \begin{cases} m & \text{if } \beta \vdash \mu \Rightarrow m \\ \text{undefined} & \text{if } \beta \vdash \mu \not\Rightarrow \end{cases}$$

The inference system made up of the inference rules (1)–(6) is monogenic; thus the definition of *Eval* makes sense.

Similarly, one can define rules that allow one to infer conclusions of the form $\beta \vdash \theta \Rightarrow \lambda\rho : \Sigma.\langle s, \varphi\rangle$, meaning that in the realization environment β, the value of θ is $\lambda\rho : \Sigma.\langle s, \varphi\rangle$. These rules are also monogenic and so give rise to a function $\mathit{Eval} : \mathrm{RealizationEnv} \to \mathrm{Functor} \rightharpoonup \mathrm{Functor}$.

2.7 Elaboration of realization expressions

To evaluate stamp expressions of the form $\mathtt{get}_S(\varphi, sp)$ that involve realization expressions, it may first be necessary to reduce the realization expression φ to a simpler form such that the rules for evaluating the stamp expression apply. The rules in this section show how to perform this reduction.

The two most interesting forms of realization expressions are $\mathsf{app}(\theta, \varphi)$, for functor application, and $\mathtt{new}\ N\,.\,\varphi$ for generativity. To handle generativity, the inference rules extend a store of currently used structure stamps each time a new stamp is picked. Thus the conclusions of the elaboration rules take the form $N, \beta \vdash \varphi \Rightarrow \varphi', N'$, and we shall always have $N' \supseteq N$.

Elaboration of functor applications involves substitution. A *substitution* is a finite map from realization variables to realization expressions. It can be represented by a realization environment β; conversely, every realization environment β represents a substitution. Application of a substitution β to a term t is written $t[\beta]$.

Structure Realizations $\boxed{\boldsymbol{N, \beta \vdash \varphi \Rightarrow \varphi', N'}}$

$$\frac{N, \beta \vdash \theta \Rightarrow \lambda\rho : \Sigma.\langle s, \varphi\rangle \quad N, \beta \vdash \varphi[\rho{=}\varphi_a \downarrow \Sigma] \Rightarrow \varphi', N'}{N, \beta \vdash \mathsf{app}(\theta, \varphi_a) \Rightarrow \varphi', N'} \tag{7}$$

$$\frac{M \cap N = \emptyset \qquad N \cup M, \beta \vdash \varphi \Rightarrow \varphi', N'}{N, \beta \vdash \mathtt{new}\ \ M\,.\,\varphi \Rightarrow \varphi', N'} \tag{8}$$

Comment: The side-condition $M \cap N = \emptyset$ forces the stamps in M to be new. By α-conversion, M can always be chosen to satisfy the side-condition.

$$\frac{N, \beta \vdash \beta_1 \Rightarrow \beta_1', N_1 \quad N_1, \beta \vdash \varphi[\beta_1'] \Rightarrow \varphi', N_2}{N, \beta \vdash \mathtt{let}\ \ \beta_1\ \ \mathtt{in}\ \ \varphi \Rightarrow \varphi', N_2} \tag{9}$$

$$\frac{N, \beta \vdash \varphi \Rightarrow \varphi', N'}{N, \beta \vdash (\varphi \downarrow \Sigma) \Rightarrow (\varphi' \downarrow \Sigma), N'} \tag{10}$$

Realization environments $\boxed{\boldsymbol{N, \beta \vdash \beta \Rightarrow \beta, N}}$

$$\frac{}{N, \beta \vdash \{\} \Rightarrow \{\}, N} \tag{11}$$

$$\frac{N, \beta \vdash \beta_1 \Rightarrow \beta_1', N_1 \quad N_1, \beta \vdash \varphi_2[\beta_1'] \Rightarrow \varphi_2', N'}{N, \beta \vdash (\beta_1, \rho{=}\varphi_2) \Rightarrow (\beta_1', \rho{=}\varphi_2'), N'} \tag{12}$$

Rule (7) deserves some explanation. The functor $\lambda\rho : \Sigma.\langle s, \varphi\rangle$ may contain free realization variables. These can be looked up in β during the elaboration of the second premise. This may seem odd in a statically scoped language, as it looks like the rule uses "dynamic binding" (β is the "call-site" environment). However, the semantics is organized in such a way that the semantic objects found for the free variables of the functor in the realization environment β at the call site are identical to the objects which were in the realization environment when the functor was declared. This is achieved, in part, by using explicit substitutions in rules (9) and (12).

3 A skeletal programming language

In this section we present a grammar and a static semantics for the skeletal language.

3.1 Grammar for programs

The grammar defining structure expressions (*strexp*) and structure-level declarations (*strdec*) is given below. (A grammar of signature expressions (*sigexp*) and specifications (*spec*) may be found in [14]).

strexp	::=	`struct` *strdec* `end`	generative
	|	*strexp*`/`*strid*	structure selection
	|	*strid*	structure identifier
	|	*strexp*`:`*sigexp*	signature constraint
	|	*fp*`(`*strexp*`)`	functor application
strdec	::=	`structure` *strid* `=` *strexp*	structure declaration
	|	`functor` *funid*`(`*strid*`:`*sigexp*`)` `=` *strexp*	functor declaration
	|		empty
	|	*strdec*`;` *strdec*	sequential

3.2 Structures and environments

An *environment*, E, is a pair (FE, SE), where FE is a *functor environment*, *i.e.* a finite map from FunId to terms θ representing static functors, while SE is a *structure environment*, *i.e.*, a finite map from StrId to structures. Concatenation of environments, written $E_1 + E_2$ is defined in the usual way.

3.3 Structure matching

Informally speaking, a structure matches a signature if it has at least the functors and structures specified in the signature and satisfies the sharing prescribed by the signature.

Formally, let N be a stamp set, β a realization environment, and $S = \langle s, \varphi \rangle$ a structure and let $\Sigma = \langle s', R, \sigma, \delta\rho.\Phi \rangle$ be a signature. We say that *S matches Σ in N and β*, written $N, \beta \vdash S$ matches Σ, if

1. $s' \subseteq s$;
2. For all $sp_1, sp_2 \in s'$, if $sp_1 \; R \; sp_2$ then $Eval(\beta)(\mathtt{get}_S(\varphi, sp_1)) = Eval(\beta)(\mathtt{get}_S(\varphi, sp_2))$, *i.e.*, they both exist and are equal stamps;
3. For all $sp \in s'$, if $[sp] \in \mathrm{Dom}(\sigma)$ then $Eval(\beta)(\sigma(sp)) = Eval(\beta)(\mathtt{get}_S(\varphi, sp))$, *i.e.*, they both exist and are equal stamps;
4. For all $fp \in s'$, if we let $\theta = Eval(\beta)(\mathtt{get}_F(\varphi, fp))$ and $\Xi = \Phi(fp)$, we have $N, \beta \vdash \theta$ matches $\Xi[\rho{=}\varphi]$;

The matching operation in item 4 is defined in Section 3.4. One of the requirements on signatures is that the only free occurrences of ρ in Ξ are of the form $\mathtt{get}_S(\rho, sp)$, where $sp \in s'$. Therefore, only the stamps of substructures of S (not functor components of S) are relevant to the substitution in item 4.

Assuming that structure S satisfies the conditions for matching the signature Σ, the structure that results from matching S with Σ is the *restriction of S to Σ*, written $restrict(S, \Sigma)$ and defined as $\langle s', \varphi \downarrow \Sigma \rangle$.

3.4 Functor matching

¿From a signature Σ it is possible to derive a so-called *free* structure, which can be thought of as a generic representative for all the structures that match Σ. This derivation can be formalized as a relation $N, \beta \vdash \mathit{Free}(\Sigma) \Rightarrow S, N'$. This relation involves a stamp set because making a free structure involves picking fresh stamps for structures which are specified in Σ. One always has $N' \supseteq N$. We omit the precise definition, for lack of space, but the process is fairly straightforward, and is justified by the Principal Signatures theorem [14].

Let $\theta = \lambda\rho_1 : \Sigma'_1.\langle s''_1, \varphi''_1\rangle$ be a functor and let $\Xi = \lambda\rho_2 : \Sigma'_2.\Sigma''_2$ be a functor signature. Write Σ'_2 in the form $\langle s'_2, R'_2, \sigma'_2, \delta\rho'_2.\Phi'_2\rangle$. We say that θ *matches* Ξ *in* N *and* β, written $N, \beta \vdash \theta$ matches Ξ, if there exist s_a, φ_a, N', φ_r and N'' such that

1. $N, \beta \vdash \mathit{Free}(\Sigma'_2) \Rightarrow \langle s_a, \varphi_a\rangle, N'$
2. $N', \beta \vdash \langle s_a, \varphi_a\rangle$ matches Σ'_1
3. $N', \beta \vdash \varphi''_1[\rho_1{=}\varphi_a \downarrow \Sigma'_1] \Rightarrow \varphi_r, N''$
4. $N'', \beta \vdash \langle s''_1, \varphi_r\rangle$ matches $\Sigma''_2[\rho_2{=}\varphi_a]$

That is, we create a free structure from Σ'_2, apply θ to it, and check that the result matches Σ''_2 after it has been instantiated with information from the free structure.

3.5 Elaboration of structure expressions

Elaboration of structure expressions is formalized in terms of a relation

$$N, \beta^d, E \vdash \mathit{strexp} \Rightarrow N_1, \beta^d_1, S, \beta^a_1$$

that consumes one realization environment, β^d and produces a structure S and *two* realization environments, β^d_1 and β^a_1. The reason is that in general we must assume that the structure expression *strexp* occurs in the body of a functor and we must achieve the effect of elaborating it formally when the functor is defined and again when the functor is applied. We introduce new realization variables to stand for all embedded functor calls, and β^d_1 maps these variables to the formal realization at "definition-time" and β^a_1 maps them to the unevaluated functor call expressions. The realization environment β^a can be regarded as "code" which is used in the functor body, which typically takes the form

$$\lambda\rho : \Sigma.\langle s, \mathtt{new}\ N.\mathtt{let}\ \beta^a\ \mathtt{in}\ \varphi_{body}\rangle$$

where N is the set of generative stamps of the functor and φ_{body} is the realization of the functor result. Details are found in rule 19.

Notation Rule 13 uses the following definitions, which relate to converting environments into structures. Let N be a stamp set, E be an environment and β a realization environment. Then functions $\mathit{combPaths}(E)$ and $\mathit{combReas}(E)$ are defined as follows. Write E in the form $(\mathit{FE}, \mathit{SE})$, where $\mathit{FE} = \{\mathit{funid}_1 \mapsto \theta_1, \ldots, \mathit{funid}_m \mapsto \theta_m\}$ and $\mathit{SE} = \{\mathit{strid}_1 \mapsto \langle s_1, \varphi_1\rangle, \ldots, \mathit{strid}_n \mapsto \langle s_n, \varphi_n\rangle\}$, for some m and n ($m, n \geq 0$). Then $\mathit{combPaths}(E) = \{\epsilon\} \cup \{\mathit{funid}_1, \ldots, \mathit{funid}_m\} \cup \bigcup_{i=1}^n \mathit{strid}_i.s_i$ (where $\mathit{strid}_i.s_i$ denotes the set of paths obtained by prepending strid_i to each path in s_i). Moreover, $\mathit{combReas}(E)$ is the view

$$\mathit{strid}_1{=}\varphi_1, \ldots, \mathit{strid}_n{=}\varphi_n, \mathit{funid}{=}\theta_1, \ldots, \mathit{funid}_m{=}\theta_m$$

Structure Expressions $\boxed{\boldsymbol{N, \beta^d, E} \vdash \mathit{strexp} \Rightarrow \boldsymbol{N_1, \beta^d_1, S, \beta^a_1}}$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \mathit{strdec} \Rightarrow N_1, \beta^d_1, E_1, \beta^a_1 \quad m \notin N_1 \\ S = \langle \mathit{combPaths}(E_1), (m, \mathit{combReas}(E_1))\rangle \end{array}}{N, \beta^d, E \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow N_1 \cup \{m\}, \beta^d_1, S, \beta^a_1} \tag{13}$$

$$\frac{N, \beta^d, E \vdash \mathit{strexp} \Rightarrow N_1, \beta^d_1, \langle s, \varphi\rangle, \beta^a_1 \quad \mathit{strid} \in s}{N, \beta^d, E \vdash \mathit{strexp/strid} \Rightarrow N_1, \beta^d_1, \langle s/\mathit{strid}, \varphi/\mathit{strid}\rangle, \beta^a_1} \tag{14}$$

$$\frac{E(\mathit{strid}) = S}{N, \beta^d, E \vdash \mathit{strid} \Rightarrow N, \{\}, S, \{\}} \tag{15}$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \mathit{strexp} \Rightarrow N_1, \beta^d_1, S, \beta^a_1 \quad N_1, \beta^d, E \vdash \mathit{sigexp} \Rightarrow \Sigma \\ N_1, \beta^d + \beta^d_1 \vdash S \text{ matches } \Sigma \quad S' = \mathit{restrict}(S, \Sigma) \end{array}}{N, \beta^d, E \vdash \mathit{strexp} : \mathit{sigexp} \Rightarrow N_1, \beta^d_1, S', \beta^a_1} \tag{16}$$

$$\frac{\begin{array}{c} E \vdash \mathit{fp} \Rightarrow \theta \quad \beta^d \vdash \theta \Rightarrow \lambda\rho_0 : \Sigma.\langle s_b, \varphi\rangle \\ N, \beta^d, E \vdash \mathit{strexp} \Rightarrow N_1, \beta^d_1, S_a, \beta^a_1 \quad N_1, \beta^d + \beta^d_1 \vdash S_a \text{ matches } \Sigma \\ S_a = \langle s_a, \varphi_a\rangle \quad N_1, \beta^d + \beta^d_1 \vdash \varphi[\rho_0{=}\varphi_a \downarrow \Sigma] \Rightarrow \varphi_b, N_2 \\ \rho' \notin \mathrm{Dom}(\beta^d + \beta^d_1) \quad \beta^d_2 = \{\rho'{=}\varphi_b\} \quad \beta^a_2 = \{\rho'{=}\,\mathsf{app}(\theta, \varphi_a)\} \end{array}}{N, \beta^d, E \vdash \mathit{fp}(\mathit{strexp}) \Rightarrow N_2, \beta^d_1 + \beta^d_2, \langle s_b, \rho'\rangle, \beta^a_1 + \beta^a_2} \tag{17}$$

Comment: The elaboration of $\varphi[\rho_0{=}\varphi_a \downarrow \Sigma]$ redoes functor applications in φ and generates fresh structures corresponding to `new`-bindings in φ.

Structure-level Declarations $\boxed{\boldsymbol{N, \beta^d, E} \vdash \mathit{strdec} \Rightarrow \boldsymbol{N_1, \beta^d_1, E_1, \beta^a_1}}$

$$\frac{N, \beta^d, E \vdash \mathit{strexp} \Rightarrow N_1, \beta^d_1, S, \beta^a_1}{N, \beta^d, E \vdash \texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \Rightarrow N_1, \beta_1, \{\mathit{strid} \mapsto S\}, \beta^a_1} \tag{18}$$

$$\frac{\begin{array}{c} N, \beta^d, E \vdash \mathit{sigexp} \Rightarrow \Sigma \\ N, \beta^d \vdash \mathit{Free}(\Sigma) \Rightarrow S_p, N_1 \quad S_p = \langle s_p, \varphi_p\rangle \\ \rho \notin \mathrm{Dom}(\beta^d) \\ N_1, (\beta^d, \rho{=}\varphi_p), E + \{\mathit{strid} \mapsto \langle s_p, \rho\rangle\} \vdash \mathit{strexp} \Rightarrow N_2, \beta^d_2, \langle s_b, \varphi_b\rangle, \beta^a_2 \\ N' = N_2 \setminus N_1 \quad \theta = \lambda\rho : \Sigma.\langle s_b, \texttt{new}\ N'\texttt{.let}\ \beta^a_2\ \texttt{in}\ \varphi_b\rangle \end{array}}{N, \beta^d, E \vdash \texttt{functor}\ \mathit{funid}\texttt{(}\mathit{strid}\texttt{:}\mathit{sigexp}\texttt{)=}\mathit{strexp} \Rightarrow N, \{\}, \{\mathit{funid} \mapsto \theta\}, \{\}} \tag{19}$$

Comment: The stamp set resulting from the elaboration of the declaration is N itself, *i.e.*, seen from outside the functor declaration, no new structures are generated. The side-condition "$\rho \notin \mathrm{Dom}(\beta^d)$" serves to distinguish the realization variable of *strid* from the realization variables of other structures, so that any free occurrence of ρ in φ_b refers to the realization of *strid*.

$$\frac{}{N, \beta^d, E \vdash \quad \Rightarrow N, \{\}, \{\}, \{\}} \tag{20}$$

$$\frac{N, \beta^d, E \vdash \mathit{strdec}_1 \Rightarrow N_1, \beta^d_1, E_1, \beta^a_1 \qquad N_1, \beta^d + \beta^d_1, E + E_1 \vdash \mathit{strdec}_2 \Rightarrow N_2, \beta^d_2, E_2, \beta^a_2}{N, \beta^d, E \vdash \mathit{strdec}_1\texttt{;}\ \mathit{strdec}_2 \Rightarrow N_2, \beta^d_1 + \beta^d_2, E_1 + E_2, \beta^a_1 + \beta^a_2} \tag{21}$$

Long functor identifiers $\boxed{\boldsymbol{E \vdash fp \Rightarrow \theta}}$

$$\frac{E(\mathit{funid}) = \theta}{E \vdash \mathit{funid} \Rightarrow \theta} \tag{22}$$

$$\frac{E(\mathit{strid}) = \langle s, \varphi \rangle}{E \vdash \mathit{strid}.\mathit{fp} \Rightarrow \mathtt{get}_F(\varphi, \mathit{fp})} \tag{23}$$

4 Conclusion

The semantics we have presented here shows that higher-order functors do not increase the complexity of the module semantics more than one would expect, and that the policy of transparent signature matching can be generalized to the higher-order case. In particular, signature matching is straightforward to check, following the definitions of the semantics.

As noted in the introduction, higher order functors behaving in accordance with this semantics have been implemented in the Standard ML of New Jersey compiler [2]. The implementation representations differ in detail from the semantic representations presented above, because of various techniques used to optimize space requirements. But taking an abstract view, there are close parallels between the semantics and the implementation.

References

1. Maria-Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Twentieth Annual ACM Symp. on Principles of Prog. Languages*, pages 465–478, New York, Jan 1993. ACM Press.
2. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, New York, August 1991. Springer-Verlag. (in press).
3. Pierre Crégut. Extensions to the sml module system. Rapport de Stage d'Ingenieur Eleve des Telecommunications, November 1992.
4. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1994. ACM Press.
5. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–354, New York, Jan 1990. ACM Press.
6. Xavier Leroy. Manifest types, modules, and separate compilation. In *Twenty First Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1994. ACM Press.
7. David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conf. on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.
8. David MacQueen. Using dependent types to express modular structure. In *Thirteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 277–286, New York, Jan 1986. ACM Press.

9. Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.
10. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.
11. John C. Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symp. on Principles of Programming Languages*, pages 28–46, New York, 1988. ACM Press.
12. Didier Rémy. Typechecking records and variants in a natural extension of ml. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 77–88, New York, Jan 1989. ACM Press.
13. Donald Sannella and Andrej Tarlecki. Extended ml: Past, present, and future. Technical Report ECS-LFCS-91-138, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.
14. Mads Tofte. Principal signatures for higher-order program modules. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press. (Extended version to appear in Journal of Functional Programming)

This article was processed using the LaTeX macro package with LLNCS style