Higher Order Modules Revisited

David MacQueen
University of Chicago

ISWIM 1966
Peter Landin

LCF ML - 1973-78
Robin Milner
Lockwood Morris
Malcolm Newey
Chris Wadsworth
Mike Gordon

Algebraic datatypes - 1969
Landin, Rod Burstall

NPL - 1977
Rod Burstall

Cardelli ML - ~1981
Luca Cardelli

Hope - 1980
Burstall, MacQueen, Sannella

Edinburgh ML - early 80s
Alan Mycroft, Kevin Mitchell

Modules for Hope - 1981
MacQueen

Standard ML 1983-1990
Milner, Burstall, MacQueen, Cardelli, Larry Paulson,
Mads Tofte, Bob Harper, MItchell, Mycroft, Scott
Guy Cousineau

Definition of Standard ML 1990
Milner, Mads Tofte, Harper

Definition of SML, Revised 1995-1997
Milner, Mads Tofte, Harper, MacQueen

SML Basis Library 1997-2004
Emden Gansner, John Reppy

Some Standard ML Implementations

- SML/NJ (1986)
  (MacQueen, Appel, Reppy, Shao, ...)

- PolyML (1985)
  (Dave Matthews)

- MLKit (1989)
  (Mads Tofte, ...)

- Moscow ML (early 1990s)
  (Romanenko, Sestoft, ...)

- MLton (1997)
  (Weeks, Fluet, ...)

- Alice ML (2002)
  (Rossberg, ...)

Some Features of SML Modules

- *Independence of interfaces and implementations*
  - a signature can be implemented by many modules
  - a module can implement (match) many signatures

- *Functors formed by abstraction with respect to structure names*
  - coherence sharing constraints for multiple parameters
  - expressed by sharing equations (deprecated) or by definitional specifications (SML 97)

- *Transparent and opaque signature ascriptions (*SML 97*)*
  - opaque ascription used for type abstraction

- *Propagation of types* can be (partially) expressed in functor signatures by sharing or definitional specifications

- Functor application is *generative*, not *applicative*

Example: coherence sharing

```
signature SA = sig type t; val f : int -> t end
signature SB = sig type s; val g : s -> bool end

(* SML 90 *)

functor F(structure A: SA; structure B: SB sharing A.t = B.s) =
struct
  val x = g(f 3)
end

(* SML 97 *)

functor F(structure A: SA; structure B: SB where type s = A.t) =
struct
  val x = g(f 3)
end
```

Variations on Modules

There are several variations on ML module system design and several approaches to formalizing these designs (notably Harper, et al -- the CMU school, and Leroy -- the Caml school).

Here I will talk about my story of modules, and in particular *strong* higher order modules as implemented in SML/NJ since 1993.

This story derives from experience with several generations of module system implementations in SML/NJ, and, by now, decades of practical use of the language.

History of Module System Implementations in SML/NJ

- 1st generation, 1987 (incomplete bootstrap version)
- 2nd generation, 1989-90 (1st order functors with sharing specs)
- 3rd generation, Feb 1993
  - full higher order functors
  - definitional specs
    (==> Harpers translucent signatures (1994) and
    Leroy's manifest types (1994))
- 4th generation, 1995-97
  - revision for compatibility with SML 97 Defn
    - drop static structure identities and sharing
    - add type (and structure) where clauses
    - entity calculus implementation of higher order functors
- 5th generation, 2010 ... (in progress, based on new semantics)

First-Order Functors in the Definition

"names": internal unique identifiers for atomic *tycons*
(primitives, datatypes, abstract types) also used as bound tycon variables

$E \in Env = (SE, TE, VE)$

$SE \in StrEnv = StrId \to Env$

$TE \in TyEnv = TycId \to Tycon$

$VE \in ValEnv = ValId \to Type$

structure: E

signature: $\Sigma = (T,E) \in Sig = NameSet * Env$ (where $T \subseteq names(E)$)

functor: funsig

funsig: $\Phi \in FunSig = NameSet * (Env * Sig)$

$\Phi = (T)(E1, (T')E2)$ – T and T' are sets of bound names (T, T' disjoint)

$\Phi = \Pi(T :E1).\Gamma(T').E2$

Functor signature instantiation

Tycon = Name (primitive)
| $\lambda\alpha$.TyExp (defined)

Realization: ϕ : Name -> Tycon (extends to Env → Env)

Sig Instantiation:
$\Sigma \geq E2$ where $\Sigma = (T)E1$
if $\exists\phi.\ \phi(E1) = E2$ and $dom(\phi) = T$

Funsig Instantiation: $\Phi = (T1)(E1, (T2)E2)$.

$\Phi \geq (E1', (T2')E2')$
if $\exists\phi.\ dom(\phi) = T1$ and $\phi(E1,(T2)E2) = (E1',(T2')E2')$

[T2 α-converted to T2' as needed to avoid free variable (name) capture]

Functor Application (Rule (54))

$$
\begin{array}{ll}
B \vdash \text{strexp} \Rightarrow E & \text{-- elaborate arg strexp to E} \\
B(\text{funid}) \geq (E1, (T2)E2) & \text{-- instantiate functor} \\
E \succcurlyeq E1 & \text{-- so that argument is matched} \\
(\text{names}(E) \cup \text{names}(B)) \cap T2 = \emptyset & \text{-- } \alpha\text{-convert to insure fresh names} \\
\hline
B \vdash \text{funid}(\text{strexp}) \Rightarrow E2 &
\end{array}
$$

Suppose: B(funid) = (Tp)(Ep,(Tr)Er) [Tp $\cap$ Tr = $\emptyset$ assumed]

The realization ϕ giving B(funid) > (E1, (T2)E2) is determined by matching E, the argument structure, with the parameter signature (Tp)Ep. This insures E $\succcurlyeq$ E1.

Rule (54) works for 1st order functors, but:
1) there is no way to extend it to handle higher order functors
2) it relies on implicit alpha conversion to model tycon generation

Why Higher Order Functors?

1. Landin's Principle of Correspondence

2. A variant of 1: Whatever entities can be defined should be definable within a module.

   - for structures, this yields hierarchical modules

   - for functors, this would yield higher-order functors

3. We use functors to factor multi-module programs. Sometimes the part of the program that we want to abstract out contains functors. [This actually happens!]

A New Static Semantics for Modules

Derived from SML/NJ implementation (4th gen)

Ideas:

1. Factoring "form" and "content" (e.g. signature/realization)

2. Static "entities" for tycons, structures, and functors (generalization and refinement of realizations ϕ)

3. An entity calculus (CBV λ-calculus with generation effects) to express functor static actions (how input tycons are mapped to output tycons, and how fresh tycons are generated)

4. Two-level elaboration of module definitions
   - direct to entities, for type checking value level
   - indirect, to entity expressions, to capture functor actions

Semantic signatures

- entity variables ρ: internal, non-shadowable variables [Harper 94] (these replace "names")

- signature representation: sig -- a mapping of identifiers to static specifications

x	$\mapsto$	(ρ, arity)	(primary tycons)
		(TycExp)	(defined tycons: λα.TypExp, relativized)
		(ρ, sig)	(structure component)
		(ρ, funsig)	(functor component)
		(Type)	(value component, relativized)

Example Signature

```
SIG =
sig
  type t
  type ‘a s = ‘a * t
  structure A : sig
    datatype v = ...
    val x : v s
  end
  val y : t -> A.v
end
```

$$
\begin{aligned}
SIG = [\ & t \mapsto (\rho_t, 0) \\
& s \mapsto \lambda\alpha.\alpha * \rho_t \\
& A \mapsto (\rho_A, [v \mapsto (\rho_v, 0) \\
& \qquad\qquad x \mapsto (\rho_v * \rho_t)]) \\
& y \mapsto \rho_t \rightarrow \rho_A.\rho_v\ ]
\end{aligned}
$$

$$SIG = ((m,n), E)$$

$$
\begin{aligned}
E = [& t \mapsto m, \\
& s \mapsto \lambda\alpha.\alpha * m \\
& A \mapsto [v \mapsto n, \\
& \qquad x \mapsto n * m] \\
& y \mapsto m \rightarrow n\ ]
\end{aligned}
$$

Example Structure matching S

```
structure S : SIG =
struct
  type t = int
  type ‘a s = ‘a * t
  structure A = struct
    datatype v = C of t
    val x = (C 3, 2)
  end
  val y = fn z => A.C(4)
end
```

Entity Environment for S:

[ρ_t = int,
  ρ_A = [ρ_v = tc_{new}]
]

Entity Expression for S:

[[ρ_t = int,
  ρ_A = [[ρ_v = new(0)]]
]]

where [[entdecls]] is the basic form of entity exp for structures.

Functor Example (old)

```
functor F(X: sig type t end) =
struct
  type u = X.t list
  datatype v = C of u
end
```

FunSig(F) = (m)(E1, (n)E2))

where E1 = [t ↦ m]

E2 = [u ↦ **list** m,
v ↦ n,
C ↦ **list** m → n]

Functor Application (old)

```
F(struct type t = int
         type s = bool end)
```

$\mathrm{FunSig}(F) = \Phi_F = (m)(E1, (n)E2))$

where $E1 = [\ t \mapsto m\ ]$

$$
\begin{aligned}
E2 = [\ & u \mapsto \text{list } m, \\
& v \mapsto n, \\
& C \mapsto \text{list } m \rightarrow n\ ]
\end{aligned}
$$

$E_{arg} = [\ t \mapsto \mathbf{int},\ s \mapsto \mathbf{bool}\ ]$

$E_{arg} \succcurlyeq E1'$ via $\phi : m \mapsto \mathbf{int}$, where $E1' = [\ t \mapsto \mathbf{int}\ ]$

$\Phi_F \geq (E1', (k)E2')$ via ϕ where

$$
\begin{aligned}
E2 = [\ & u \mapsto \textbf{list int}, \\
& v \mapsto \mathbf{k}, \\
& C \mapsto \textbf{list int} \rightarrow \mathbf{k}\ ]
\end{aligned}
$$

and **k** is a fresh name (i.e. atomic tycon).

Functor Example (new)

```
functor F(X: sig type t end) =
struct
  type u = X.t list
  datatype v = C of u
end
```

functor signature: $fsig_F = \Pi\rho{:}\ \Sigma_p.\ \Sigma_r$

$\Sigma_p = [\ t \mapsto (\rho_t, 0)\ ]$

$\Sigma_r = [\ u \mapsto \mathbf{list}(\rho_X.\rho_t),$
$\quad v \mapsto (\rho_v, 0),$
$\quad C \mapsto \mathbf{list}\ (\rho_X.\rho_t) \rightarrow \rho_v\ ]$

entity expression: $exp_F = \lambda\rho_X.\ [[\ \rho_v = new(0)\ ]]$

functor entity: $ent_F = (\ exp_F, EE_c\ )$
(where EE_c is "current" entity env)

static functor: $F = \langle\ fsig_F, ent_F\ \rangle$

Functor Application Rule (New Semantics)

$$\frac{\begin{array}{l} EE(ep) = (\lambda\rho.body,\ EE_1) \\ arg,\ EE \Downarrow R_{arg} \\ body,\ (EE_1,\ \rho \mapsto R_{arg}) \Downarrow R \end{array}}{ep(arg),\ EE \Downarrow R}$$

(simplified by omitting signature matching and coercion on argument)

Example: `F(struct type t = int end)`

$EE(F) = (\lambda\rho_X.\ [[\ \rho_v = new(0)\ ]],\ EE_1)$

$[[\ \rho_t\ = int\ ]],\ EE\ \Downarrow R_{arg}\ = ([\rho_t\ \mapsto \textbf{int}],\ EE)$

$[[\ \rho_v = new(0)\ ]],\ EE_2 \Downarrow ([\ \rho_t\ \mapsto \textbf{tc}],\ EE_2)$, where $EE_2 = EE_1,\ \rho_X \mapsto R_{arg}$

Higher Order Functor Example

`SIG = sig type t end` $(\Sigma = [\ t \mapsto (\rho_t, 0)\ ]$

```
functor Apply(F: SIG => SIG, A: SIG) = F(A)
```

FunSig for Apply:

$\Pi\rho: \Sigma p.\Sigma$ where

$\Sigma p = [\ F \mapsto (\rho_F, \Pi\rho_X{:}\Sigma.\Sigma),\ A \mapsto (\rho_A, \Sigma)\ ]$

Entity expression for Apply:

$\lambda\rho.\ \rho.\rho_F(\rho.\rho_A)$

Static functor for Apply:

$<\lambda\rho.\ \rho.\rho_F(\rho.\rho_A),\ EE_c>$ where EE_c is current entity environment

Observations and Conclusions

• The entity calculus is a very natural model for first-order functors, but once you have it, higher-order modules come for free.

• The entity calculus model is easily translated to implementation – indeed, it was derived from a pre-existing implementation!

• “Strong” or “true” higher-order functors are naturally supported, but the inherent conflict with “pure” separate compilation is made even clearer. A *complete* static signature for a functor would have to encorporate the entity function encoding the functor static action.

But lack of “pure” separate compilation has not been a practical problem for SML programmers. Adequate separate compilation is easy to achieve.