Types For Modules

Claudio Russo

November 12, 1998

Overview

Aim: Explain and *naturally* extend Standard ML Modules.

Review Standard ML type theory

- typical modules and types

Higher Order Modules: functors as components, arguments and results

- motivating example
- generalising signatures
- generalising enrichment
- incorporating generativity

First Class Modules: modules as Core values

- introducing package types
- rules
- motivating example

Assume: Basic familiarity with SML and Type Theory.

First Order Modules ...

- Structure expressions have types similar to **record** types:

```
structure Nat = struct type nat = int;
                       val z = 0;
                       fun s n = n + 1;
                       fun i b f n =
                        if n=0 then b else i b f (n-1);
                       structure Even = struct ... end
                end;
```

$: (\mathbf{nat} = \mathbf{int}, \mathbf{z} : \mathbf{int}, \mathbf{s} : \mathbf{int} \rightarrow \mathbf{int},$
$\quad \mathbf{i} : {}'a \rightarrow ({}'a \rightarrow {}'a) \rightarrow \mathbf{int} \rightarrow {}'a), \mathbf{Even} : (\ldots))$

- Signature expressions denote **families** of structure types:

```
signature NAT = struct type nat;
                       val z : nat;
                       val s : nat→nat
                       val i : 'a→('a→'a)→nat→'a
                end;
```

$\triangleright \Lambda\alpha.(\mathbf{nat} = \alpha, \mathbf{z} : \alpha, \mathbf{s} : \alpha \rightarrow \alpha, \mathbf{i} : {}'a \rightarrow ({}'a \rightarrow {}'a) \rightarrow \alpha \rightarrow {}'a)$

- Abstractions introduce **existential** types:

```
structure AbsNat = Nat :> NAT;
```

$: \ \exists\alpha.(\mathbf{nat} = \alpha, \mathbf{z} : \alpha, \mathbf{s} : \alpha \rightarrow \alpha, \mathbf{i} : {}'a \rightarrow ({}'a \rightarrow {}'a) \rightarrow \alpha \rightarrow {}'a)$

. . . First Order Modules

- Functors are **polymorphic functions** from structures . . .

```
functor A (N : NAT) =
     struct fun add n m = N.i n N.s m
     end;
```

$: \forall\alpha.\ (\mathbf{nat} = \alpha, \mathbf{z} : \alpha, \mathbf{s} : \alpha\to\alpha, \mathbf{i} : {}'a\to({}'a\to{}'a)\to\alpha\to{}'a) \to$
$\quad (\mathbf{add} : \alpha\to\alpha\to\alpha)$

- . . . to **existential** structures.

```
functor AbsFun (N : NAT) = N :> NAT;
```

$: \forall\alpha.\ (\mathbf{nat} = \alpha, \mathbf{z} : \alpha, \mathbf{s} : \alpha\to\alpha, \mathbf{i} : {}'a\to({}'a\to{}'a)\to\alpha\to{}'a) \to$
$\quad \exists\beta.(\mathbf{nat} = \beta, \mathbf{z} : \beta, \mathbf{s} : \beta\to\beta, \mathbf{i} : {}'a\to({}'a\to{}'a)\to\beta\to{}'a)$

Functor application is **generative**: *"Distinct applications of the same functor return* **distinct** *types."*

Types for Higher Order Modules

Generalise first order types:

- A module type $\mathcal{M}$ is either:
  - a structure type $\mathcal{S}$; or
  - a functor type $\mathcal{F} \equiv \forall P.\mathcal{M} \to \mathcal{M}'$.
- Structure types $\mathcal{S}$ can specify module components $\mathrm{X} : \mathcal{M}, \mathcal{S}$.
- Because of generativity, module expressions have existential module type $\exists P.\mathcal{M}$.

$$
\begin{array}{lcll}
\mathcal{M} & ::= & \mathcal{S} & \text{structure} \\
 & | & \mathcal{F} & \text{functor} \\
 & & & \\
\mathcal{S} & ::= & \mathrm{t} = \tau, \mathcal{S}' & \\
 & | & \mathrm{x} : v, \mathcal{S}' & \\
 & | & \mathrm{X} : \mathcal{M}, \mathcal{S}' & \text{module spec} \\
 & | & \epsilon_{\mathcal{S}} & \\
\mathcal{F} & ::= & \forall P.\mathcal{M} \to \mathcal{M}' & \text{functor} \\
 & & & \\
\mathcal{X} & ::= & \exists P.\mathcal{M} & \text{existential module} \\
 & & & \\
\mathcal{L} & ::= & \Lambda P.\mathcal{M} & \text{signature}
\end{array}
$$

An Example: Evaluating Polynomials

The specification of a module evaluating polynomials.

```
signature POLY =
 sig module Nat : NAT
     val eval : Nat.nat→(list Nat.nat)→Nat.nat
 end
```

Intention:

$$\text{eval}\ \ x\ [a_0, a_1, \ldots\ , a_n] = a_0 x^0 + a_1 x^1 + \cdots + a_n x^n.$$

. . . Evaluating Polynomials

```
module MkPoly =
functor(N:NAT)
functor(A:funsig(N:NAT with nat = N.nat)
          sig val add:N.nat→N.nat→N.nat end)
functor(M:funsig(N:NAT with nat = N.nat)
          funsig(A:funsig(N:NAT with nat = N.nat )
                   sig val add:N.nat→N.nat→N.nat end))
          sig val mult:N.nat→N.nat→N.nat  end)
  struct
    module Nat = N
    module Add = A N
    module Mult = M N A
    fun eval x [] = Nat.z
      | eval x (h::l) =
             Add.add h (Mult.mult x (eval x l)))
  end
```

`MkPoly` assumes:

`N` is a structure implementing `NAT`.

`A` is functor from `N` to addition.

`M` is a higher-order functor mapping `N` and `A` to multiplication.

Motivating Functor Enrichment . . .

```
module A =
   functor(N:sig type nat = Nat.nat
                 val z: nat
                 val s: nat→nat
                 val i: 'a→('a→'a)→nat→'a
             end)
      struct fun add n m = N.i n N.s m end
```

Expect `MkPoly Nat A` **to typecheck.**

. . . Motivating Functor Enrichment

```
module A' =
   functor(N:sig type nat
                 val s: nat→nat
                 val i: nat→(nat→nat)→nat→nat
             end)
      struct fun add n m = N.i n N.s m
             fun sum b [] = b
               | sum b (h::l) = add h (sum b l)
      end
```

Functor `A'` is "more general" than `A`:

- more polymorphic
- expects smaller argument
- produces bigger result

We also want `MkPoly Nat A'` to typecheck.

Generalising Signatures ...

In Standard ML, signatures denote **families of structures**, e.g.:

$$\ldots \vdash \textbf{sig type t end} \triangleright \Lambda\alpha.(\mathbf{t} = \alpha)$$

What about the functor signature:

$$\ldots \vdash \textbf{funsig}(\mathbf{X}\text{:}\textbf{sig type t end})\textbf{sig type u end} \triangleright ?$$

Can we use it to specify a **family of functors**?

Idea:

$$\ldots[\mathbf{X} : \mathbf{t} = \alpha] \vdash \textbf{sig type u end} \triangleright \Lambda\beta.(\mathbf{u} = \beta),$$

we can use the result type β to define the family of functors:

$$\Lambda\beta.\forall\alpha.(\mathbf{t} = \alpha) \to (\mathbf{u} = \beta). \tag{X}$$

"The family of functors that are polymorphic in $\mathbf{t}$, indexed by the type $\mathbf{u}$."

... Generalising Signatures ...

$$\Lambda\beta.\forall\alpha.(\mathbf{t} = \alpha) \rightarrow (\mathbf{u} = \beta). \qquad \text{(X)}$$

- Then

  $$\mathbf{functor(X : sig\ type\ t\ end)struct\ type\ u = int\ end}$$

  has type:

  $$\forall\alpha.(\mathbf{t} = \alpha) \rightarrow (\mathbf{u} = \mathbf{int})$$

  which matches (X) by choosing $[\mathbf{int}/\beta]$.

- But

  $$\mathbf{functor(X : sig\ type\ t\ end)struct\ type\ u = X.t\ end}$$

  has type:

  $$\forall\alpha.(\mathbf{t} = \alpha) \rightarrow (\mathbf{u} = \alpha)$$

  which can't ever match (X) since α is in the scope of β.

Observation: The definition of $\mathbf{u}$ is a function of $\mathbf{t}$.
Conclusion: We should make β a function of α.

... Generalising Signatures ...

Sandip Biswas's Idea (POPL'95):

Refinement: we raise β to higher-order to capture the dependency on α:

$$\ldots \vdash \textbf{funsig}(\textbf{X}{:}\textbf{sig type t end})\textbf{sig type u end} \rhd$$
$$\Lambda\beta.\forall\alpha.(\mathbf{t} = \alpha) \to (\mathbf{u} = \beta\ \alpha) \qquad (*)$$

Intuition: $\mathbf{u}$ is a function (namely β) of $\mathbf{t}$ (a.k.a. α).

"The family of functors that are polymorphic in $\mathbf{t}$, indexed by the functional dependency of $\mathbf{u}$ on $\mathbf{t}$."

- $\textbf{functor}(\textbf{X} : \textbf{sig type t end})\textbf{struct type u} = \textbf{int end}$ has type:
  $$\forall\alpha.(\mathbf{t} = \alpha) \to (\mathbf{u} = \mathbf{int})$$
  still matches $(*)$ by choosing $[\Lambda\alpha.\mathbf{int}/\beta]$.

- $\textbf{functor}(\textbf{X} : \textbf{sig type t end})\textbf{struct type u} = \textbf{X.t end}$ has type:
  $$\forall\alpha.(\mathbf{t} = \alpha) \to (\mathbf{u} = \alpha)$$
  which now matches $(*)$ by choosing $[\Lambda\alpha.\alpha/\beta]$.

. . . Generalising Signatures

The general rule is:

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{S}' \rhd \Lambda Q.\mathcal{M}'}{\mathcal{C} \vdash \mathbf{funsig}(\mathrm{X}{:}\mathrm{S})\mathrm{S}' \rhd \Lambda \hat{Q}.\forall P.\mathcal{M} \to [\hat{Q}/Q]\,(\mathcal{M}')}$$

where $\hat{Q}$ is higher-order variant of Q,

$$P = \{\alpha_1, \ldots, \alpha_n\}$$

and

$$[\hat{Q}/Q] = \{\beta \mapsto \beta\ \alpha_1 \cdots \alpha_n | \beta \in Q\}.$$

Generalising Enrichment ...

In SML, a functor $\mathbf{F}$ may be applied to any argument that **enriches** ($\_ \succeq \_$) a **realisation** (φ) of its formal argument.

$$\frac{\mathcal{C}(\mathrm{F}) = \forall P.\mathcal{S}' \rightarrow \exists Q.\mathcal{S} \qquad \mathcal{C} \vdash \mathrm{s} : \mathcal{S}'' \qquad \boxed{\mathcal{S}'' \succeq \varphi\left(\mathcal{S}'\right)} \qquad \mathrm{Dom}(\varphi) = P}{\mathcal{C} \vdash \mathrm{F}\ \mathrm{s} : \exists Q.\varphi\left(\mathcal{S}\right)} \qquad \text{(E)}$$

Here "$\mathcal{S}$ **enriches** $\mathcal{S}'$" is defined as:

$$\frac{\begin{array}{l} \mathrm{Dom}(\mathcal{S}) \supseteq \mathrm{Dom}(\mathcal{S}') \\ \forall \mathrm{t} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \\ \forall \mathrm{x} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{x}) \succeq \mathcal{S}'(\mathrm{x}) \\ \forall \mathrm{X} \in \mathrm{Dom}(\mathcal{S}').\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X}) \end{array}}{\mathcal{S} \succeq \mathcal{S}'}$$

- $\mathcal{S}$ must have at least the components of $\mathcal{S}'$.
- common types are equal.
- common values may be more polymorphic in $\mathcal{S}$.
- common structures may be richer in $\mathcal{S}$.

In Higher-Order Modules, functors may take functors as arguments ...

... Generalising Enrichment

Q: How do we define $\mathcal{F} \succeq \mathcal{F}'$

Intuition: First try to define a notion of **functor instance**:

$$\boxed{\forall P.\mathcal{M}_P \to \mathcal{M}'_P > \mathcal{M} \to \mathcal{M}'}$$

$$\frac{\mathcal{M} \succeq \varphi\left(\mathcal{M}_P\right) \quad \varphi\left(\mathcal{M}'_P\right) \succeq \mathcal{M}' \quad \mathrm{Dom}(\varphi) = P}{\forall P.\mathcal{M}_P \to \mathcal{M}'_P > \mathcal{M} \to \mathcal{M}'}$$

"**polymorphic instantiation & contra-variant enrichment**"

Then try to define:
$\mathcal{F} \succeq \mathcal{F}'$ iff "every instance of $\mathcal{F}'$ is an instance of $\mathcal{F}$"

$$\frac{\forall \mathcal{M}, \mathcal{M}'. \quad (\mathcal{F}' > \mathcal{M} \to \mathcal{M}') \supset (\mathcal{F} > \mathcal{M} \to \mathcal{M}')}{\mathcal{F} \succeq \mathcal{F}'}$$

(Strictly speaking, this is not a definition but it can be turned into one!).

$\_ \succeq \_$ is a **pre-order**. Suitable for **subtyping**!

Incorporating Generativity ...

Q: What about generativity? Can't functors return new types?

In Standard ML functors have the form:

$$\forall P.\mathcal{S} \to \boxed{\exists Q.\mathcal{S}'}.$$

Q is the set of **generative types**.

$$\begin{array}{l}\textbf{functor}(\mathbf{X} : \textbf{sig type t end}) \\ \qquad (\textbf{struct type u} = \mathbf{X}.\textbf{t end} \setminus \textbf{sig type u end}) \\ \qquad\qquad\qquad : \forall\alpha.(\mathbf{t} = \alpha) \to \exists\beta.(\mathbf{u} = \beta)\end{array}$$

We've assumed functors have the form:

$$\forall P.\mathcal{M} \to \boxed{\mathcal{M}'}.$$

What happened to "$\exists Q.$"?

The point is, we don't need these generative sets ...

... Incorporating Generativity

... if we adopt **applicative functors**!

Idea: replace the "generative" rule:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \exists Q.\mathcal{M}'}{\mathcal{C} \vdash \mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} : \forall P.\mathcal{M} \to \exists Q.\mathcal{M}'}$$

by the **applicative** rule:

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M} \quad \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{m} : \exists Q.\mathcal{M}'}{\mathcal{C} \vdash \mathbf{functor}(\mathrm{X} : \mathrm{S})\mathrm{m} : \exists \hat{Q}.\forall P.\mathcal{M} \to [\hat{Q}/Q]\,(\mathcal{M}')}$$

The substitution $[\hat{Q}/Q]$ **parameterises** each existential variable in Q by every type parameter in P, tracking type dependencies.

This is a **skolemisation** step that is necessary for **type soundness**.

With this change, functors have the desired form $\forall P.\mathcal{M} \to \mathcal{M}'$.

Functor application is **applicative**: *"Distinct applications of the same functor to equivalent types return equivalent types."*

First Class Modules

The Core language expresses **algorithms** and **data structures**.

The Modules language expresses **architecture**.

Existing Core/Modules **stratification** prevents the construction of modules using computational mechanisms of the Core.

"No runtime reconfiguration of Modular programs."

Goal: Relax the stratification.

Previous approach: coalesce Core and Modules into one language [cf. Harper, Lillibridge, Stone].

Our approach:

- maintain the distinction between Core and Modules.
- encapsulate module values in a new type of core value.

Package Types

Idea: Extend Core types with the **package type** $<\mathrm{S}>$.

A Core value of type

$$<\mathrm{S}>$$

encapsulates a module value that matches signature S.

The Core expression

$$\mathbf{pack}\ \mathrm{m}\ \mathbf{as}\ \mathrm{S}$$

encapsulate the module value of m in a Core value of package type $<\mathrm{S}>$.

The Core expression

$$\mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \mathrm{X} : \mathrm{S}\ \mathbf{in}\ \mathrm{e}'$$

eliminates a value of package type $<\mathrm{S}>$.

"Evaluate the Core expression e *to an encapsulated module value, bind this value to* X*, then evaluate the Core expression* e'*".*

Typing Rules

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{M}}{\mathcal{C} \vdash <\mathrm{S}> \triangleright <\exists P.\mathcal{M}>}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{m} : \exists P.\mathcal{M} \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{M}' \\ \mathcal{M} \succeq \varphi\left(\mathcal{M}'\right) \\ \mathrm{Dom}(\varphi) = Q\end{array}}{\mathcal{C} \vdash \mathbf{pack}\ \mathrm{m}\ \mathbf{as}\ \mathrm{S} : <\exists Q.\mathcal{M}'>}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathrm{e} : <\exists Q.\mathcal{M}> \\ \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{M} \\ Q \cap \mathrm{FV}(\mathcal{C}) = \emptyset \\ \mathcal{C}[\mathrm{X} : \mathcal{M}] \vdash \mathrm{e}' : u \\ Q \cap \mathrm{FV}(u) = \emptyset\end{array}}{\mathcal{C} \vdash \mathbf{open}\ \mathrm{e}\ \mathbf{as}\ \mathrm{X} : \mathrm{S}\ \mathbf{in}\ \mathrm{e}' : u}$$

Analogous to the existential introduction and elimination rules of Type Theory!

An Example: Dynamically Sized Arrays . . .

The specification of a structure implementing fixed-size arrays:

```
signature Array =
 sig type 'a array;
     val init: 'a → 'a array;
     val sub: 'a array → int → 'a;
     val update : 'a array → int → 'a  → 'a array
 end;
```

Intention:

Implement a function array n that dynamically returns an implementation of arrays of size 2^n, $(n \geq 0)$.

Side Condition: operations sub and update should have worst-case time complexity $O(n)$ (ie. logarithmic in $|array|$).

. . . Dynamically Sized Arrays . . .

Base Case: structure `ArrayZero` implements arrays of size 2^0:

```
module ArrayZero =
  struct
     type 'a array = 'a;
     val init = λx.x;
     val sub = λa.λi.a;
     val update = λa.λi.λx.x
  end;
```

. . . Dynamically Sized Arrays . . .

Inductive Step: functor `ArraySucc` mapping an implementation of arrays of size 2^n to an implementation of arrays of size 2^{n+1}:

```
module ArraySucc =
   functor(A:Array)
   struct
       type 'a array = 'a A.array * 'a A.array;
       val init = λx. (A.init x, A.init x)
       val sub =
           λ(fst,snd).λi.
              if even i
              then A.sub fst (i/2)
              else A.sub snd (i/2);
       val update =
           λ(fst,snd).λi.λx.
              if even i
              then (A.update fst (i/2) x, snd)
              else (fst, A.update snd (i/2) x)
   end;
```

. . . Dynamically Sized Arrays

Inductive Definition:

```
fun array = λn.
              if (n = 0)
              then pack ArrayZero as Array
              else open array (n-1) as A:Array in
                   pack ArraySucc(A) as Array;
```

The function array has type int $\rightarrow$ <Array>.

The application array n returns an abstract implementation of arrays of size 2^n.

Summary & Conclusions

First-Order Modules

- structure types like record types
- signature denote parameterised types
- abstractions introduce existential types
- functors are polymorphic functions
- enrichment is subtyping
- **no evidence of dependent types!** [cf. MacQueen, Mitchell, Harper].

Higher-Order Modules

- A natural generalisation of first-order syntax and types.
- higher-order type variables capture type dependencies.
- functor enrichment combines polymorphic generalisation with contravariant enrichment
- retain generativity via applicative functors
- **no need to abandon the Definition!**

First-Class Modules

- package types encapsulate Module values as Core Values.
- just need three constructs and rules
- similar to existential types
- strictly more expressive than stratified calculus.
- **no need to abandon the Definition!**