Types For Higher Order Modules

Claudio V. Russo

May 26, 1997

Overview

Aim: Generalise Standard ML modules to higher-order.

- functors as structure components
- functors as arguments
- functors as results

Outline:

- motivating example
- generalising signatures
- generalising enrichment
- incorporating generativity
- an algorithm for signature matching
- conclusions

Assume: familiarity with SML.

An Example

The specification of a module evaluating polynomials.

```
signature NAT =
   sig type nat
       val z:nat
       val s:nat->nat
       val i: !'a.'a->('a->'a)->nat->'a
   end
signature POLY =
   sig module Nat:NAT
       val eval:Nat.nat->(list Nat.nat)->Nat.nat
   end
```

Intention:

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

A Top-Down Design with a Higher-Order Functor

```
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 N = ...

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

Expect `MkPoly N 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 val 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 N A'` **to typecheck.**

. . . An efficient Implementation

```
module N = struct type nat = int
                  val z=0
                  val s n = n+1
                  fun i b f 0 = b
                    | i b f n = f (i b f (n-1))
           end
module A = functor(N:sig end)
           struct val add i j = i+j
           end
module M = functor(N:sig end)
           functor(A:funsig(N:NAT with nat = int)
                     sig end)
           struct val mult i j = i*j
           end
```

Fast but insecure (no abstraction).

An Abstract & Efficient Implementation.

```
module FastNat =
 struct
    module N = ...
    module A = ...
    module M = ...
 end :>
 sig
     module N: NAT
     module A: funsig(N:NAT with nat = N.nat)
                sig val add: N.nat->N.nat->N.nat end
     module M: funsig(N:NAT with nat = N.nat)
               funsig(A:funsig(N:NAT with nat = N.nat)
                         sig end)
                sig val mult: N.nat->N.nat->N.nat end
 end
```

`FastNat.N.nat` **is now abstract.**
`MkPoly FastNat.N FastNat.A FastNat.M` **type checks.**

We need functor components to do this abstraction.

"Types" of (First Order) Modules ...

- Structure expressions are classified by structures (like records types):

```
structure Y = struct type t = int
                     structure X : struct type u = bool
                                          val x = true
                                   end
                     val y = (1,X.x)
              end
```

$$\mathsf{Y} : (\ \mathbf{t} = \mathbf{int},\ \mathbf{X} : (\mathbf{u} = \mathbf{bool},\ \mathbf{x} : \mathbf{bool}),\ \mathbf{y} : \mathbf{int} \times \mathbf{bool})$$

- Signature expressions denote **families** of structures:

```
signature S = sig type t
                  structure X : sig type u
                                    val x: u
                                end
                  val y : t × X.u
              end
```

$$\mathsf{S} \triangleright \Lambda\alpha,\beta.\ (\mathbf{t} = \alpha,\ \mathbf{X} : (\ \mathbf{u} = \beta,\ \mathbf{x} : \beta),\ \mathbf{y} : \alpha \times \beta)$$

- Abstractions introduce existential types:

$$(\mathtt{Y :> S})\quad :\quad \exists\alpha,\beta.\ (\mathbf{t} = \alpha,\ \mathbf{X} : (\mathbf{u} = \beta,\ \mathbf{x} : \beta),\ \mathbf{y} : \alpha \times \beta)$$

. . . "Types" of (First-Order) Modules

- Functors are polymorphic maps from structures . . .

  `functor` F (Y : sig type t val y:t end) = Y

  F : $\forall\alpha.\ (\mathbf{t} = \alpha,\mathbf{y} : \alpha) \rightarrow (\mathbf{t} = \alpha,\mathbf{y} : \alpha)$

- . . . to existential structures (generativity).

  `functor` G(Y : sig type t val y : t end) =
(Y :> sig type t val y : t end)

  G : $\forall\alpha.\ (\mathbf{t} = \alpha,\mathbf{y} : \alpha) \rightarrow \exists\beta.(\mathbf{t} = \beta,\mathbf{y} : \beta)$

“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 $\exists P.\mathcal{M}$.

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

Generalising Signatures ...

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

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

What about the functor signature

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

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

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

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, indexed by the result type $\mathbf{u}$, which are polymorphic in $\mathbf{t}$."

... Generalising Signatures ...

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

- Then

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

  has type:

  $$\forall\alpha.\mathbf{t} = \alpha \to \mathbf{u} = \mathbf{int}$$

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

- But

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

  has type:

  $$\forall\alpha.\mathbf{t} = \alpha \to \mathbf{u} = \alpha$$

  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:

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

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

"The family of functors, indexed by the argument-result dependency of $\mathbf{u}$ on $\mathbf{t}$, which are polymorphic in $\mathbf{t}$."

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

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

... Generalising Signatures ...

An Example:

The **higher-order** functor:

$$\begin{array}{c}\mathbf{functor(G : funsig(X : sig\ type\ t\ end)sig\ type\ t\ end)}\\ \mathbf{functor(Y : sig\ type\ t\ end)G\ (G\ Y).}\end{array}$$

has module type:

$$\begin{array}{l}\forall\beta.(\forall\alpha.\mathbf{t} = \alpha \to \mathbf{t} = \beta\ \alpha) \to\\ \qquad\qquad \forall\delta.(\mathbf{t} = \delta) \to (\mathbf{t} = \beta\ (\beta\ \delta)).\end{array}$$

Note:

- The functor is polymorphic in $\mathbf{G}$'s argument-result dependency β.
- $\mathbf{G}$ is polymorphic in α.
- $\mathbf{G}$'s polymorphism is exploited twice.
- The functor returns a polymorphic functor.

. . . Generalising Signatures

The general rule is:

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

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 which is **at least as rich** ($\_ \succeq \_$) as an **instance** of its formal argument.

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

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 $\mathcal{F}$"

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

But this isn't a definition . . .

- The relation $\succeq$ occurs **negatively** in its "definition".

... Generalising Enrichment

Instead, we can define $\boxed{\mathcal{F} \succeq \mathcal{F}'}$:

$$\frac{\begin{array}{ll}\mathcal{M}_Q \succeq \varphi\,(\mathcal{M}_P) & \varphi\left(\mathcal{M}'_P\right) \succeq \mathcal{M}'_Q \\ \mathrm{Dom}(\varphi) = P & Q \cap \mathrm{FV}(\forall P.\mathcal{M}_P \to \mathcal{M}'_P)\end{array}}{\forall P.\mathcal{M}_P \to \mathcal{M}'_P \succeq \forall Q.\mathcal{M}_Q \to \mathcal{M}'_Q}$$

Informally:

$$\mathcal{F} \succeq \mathcal{F}'$$

if and only if

"the **generic instance** of $\mathcal{F}'$ is an instance of $\mathcal{F}$":

Observe:

- This is a **syntactic** condition.
- It yields a proper **inductive definition**.

We can show:

- $\_ \succeq \_$ is a **pre-order**. Good for subtyping!
- $\_ \succeq \_$ satisfies the **properties** of the previous slide.

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**.

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

We've assumed functors have the form:

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

What happened to Q?

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

... Incorporating Generativity ...

We can get away with something simpler:

applicative functors!

Idea: replace the "generative" rules:

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

by the "applicative" rules:

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

Functors now have the form $\forall P.\mathcal{M} \to \mathcal{M}'$.

. . . Incorporating Generativity . . .

Unfortunately, rule $(*)$ is not sound. **Counterexample**:

$$\begin{array}{l}\textbf{functor}(\textbf{X} : \textbf{sig type t end}) \\ \qquad (\textbf{struct type u} = \textbf{X.t end} :> \textbf{sig type u end})\end{array}$$

$$: \exists\beta.\forall\alpha.\mathbf{t} = \alpha \to \mathbf{u} = \beta \qquad \text{(X)}$$

Observe: The definition of $\mathbf{u}$ is a function of $\mathbf{t}$.

Type Insecurity: β doesn't capture the dependency on α.

Idea: Skolemise existential variables! E.g.:

$$\begin{array}{l}\textbf{functor}(\textbf{X} : \textbf{sig type t end}) \\ \qquad (\textbf{struct type u} = \textbf{X.t end} :> \textbf{sig type u end})\end{array}$$

$$: \exists\beta.\forall\alpha.\mathbf{t} = \alpha \to \mathbf{u} = \beta\ \alpha$$

Now the definition of $\mathbf{u}$ is cleary a function (namely β!) of $\mathbf{t}$ (a.k.a. α).

... Incorporating Generativity ...

In general, the correct applicative rule is:

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

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

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

and

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

$[\hat{Q}/Q]$ takes care of the skolemisation.

... Incorporating Generativity ...

Functor elimination is simplified to:

$$\frac{\begin{array}{ll}\mathcal{C}(\mathrm{F}) = \forall P.\mathcal{M}' \to \mathcal{M} & \mathcal{C} \vdash \mathrm{m} : \exists Q.\mathcal{M}'' \\ \mathcal{M}'' \succeq \varphi\left(\mathcal{M}'\right) & \mathrm{Dom}(\varphi) = P\end{array}}{\mathcal{C} \vdash \mathrm{F}\ \mathrm{m} : \exists Q.\varphi\left(\mathcal{M}\right)} \tag{E'}$$

Signature Matching ...

Recall:

Structure $\mathcal{S}$ **matches** signature $\Lambda P.\mathcal{S}'$
if and only if
$\mathcal{S} \succeq \varphi\left(\mathcal{S}'\right)$, where $\mathrm{Dom}(\varphi) = P$.

As in Standard ML, we need to "guess" matching realisations in the rules for $\mathrm{m}\ \mathrm{m}'$, $\mathrm{m} : \mathrm{S}$, $\mathrm{m} : >\mathrm{S}$.

Even worse: to check $\mathcal{F} \succeq \mathcal{F}'$ we already need to guess a matching realisation:

$$\frac{\boxed{\mathcal{M}_Q \succeq \varphi\left(\mathcal{M}_P\right)} \qquad \varphi\left(\mathcal{M}'_P\right) \succeq \mathcal{M}'_Q \qquad \mathrm{Dom}(\varphi) = P \qquad Q \cap \mathrm{FV}(\forall P.\mathcal{M}_P \to \mathcal{M}'_P)}{\forall P.\mathcal{M}_P \to \mathcal{M}'_P \succeq \forall Q.\mathcal{M}_Q \to \mathcal{M}'_Q}$$

Worry: Realisations are **higher-order** substitutions!
Higer-order matching is undecidable and non-unitary.
Do we still have a type inference algorithm?

. . . Signature Matching . . .

Nae Worries! The matching problems we have to solve are in fact very easy [Biswas95]. They are related to Dale Miller's "patterns".

Fact: There is an algorithm

$$\boxed{\forall P.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \ \downarrow \ \_}$$

for computing matching realisations [Biswas95].

Theorem: Provided . . . then

$$\begin{array}{c} \forall P.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \ \downarrow \ \varphi \\ \text{iff} \\ \mathcal{M} \succeq \varphi\left(\mathcal{M}'\right). \end{array}$$

The Signature Matching Algorithm ...

Observe: Because of functors, the **definitions of enrichment and matching are now interwined**.

Consequence: The algorithm needs to simultaneously compute a realisation and check enrichment.

Given module type $\mathcal{M}$ and signature $\Lambda Q.\mathcal{M}'$

$$\forall P.\forall R \vdash \mathcal{M} \succeq \mathcal{M}' \downarrow \_$$

computes a realisation

$$\varphi$$

with

- $\mathcal{M} \succeq \varphi\,(\mathcal{M}')$
- $\mathrm{Dom}(\varphi) = Q$
- $\mathrm{FV}(\mathrm{Rng}(\varphi)) \subseteq P$
- $\mathrm{FV}(\mathrm{Rng}(\varphi)) \cap R = \emptyset$

- R: Restricted variables (arguments to patterns). Records type parameters of functors. Grows as we enter functors $\forall R'. \ldots$.
- P: free variables (constants).

. . . The Signature Matching Algorithm . . .

$$\boxed{\forall P.\forall R \vdash \mathcal{S} \succeq \mathcal{S}' \;\downarrow\; \_}$$

$$\frac{\begin{array}{l} \alpha \notin P \cup R \\ \mathrm{t} \in \mathrm{Dom}(\mathcal{S}) \\ \mathrm{FV}(\mathcal{S}(\mathrm{t})) \cap R \subseteq \{\beta_i | i \in [n]\} \\ \forall P.\forall R \vdash \mathcal{S} \succeq [\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha]\left(\mathcal{S}'\right) \;\downarrow\; \varphi \end{array}}{\forall P.\forall R \vdash \mathcal{S} \succeq \mathrm{t} = \alpha\ \beta_0 \cdots \beta_{n-1}, \mathcal{S}' \;\downarrow\; ([\Lambda\beta_0 \cdots \beta_{n-1}.\mathcal{S}(\mathrm{t})/\alpha] \mid \varphi)}$$

- α is a (flexible) variable in the signature.
- β_i are distinct parameters from R.
- $\alpha\ \beta_0 \cdots \beta_{n-1}$ is a Dale Miller "pattern".
- the realisation for α is applied to $\mathcal{S}$', before computing φ.

... The Signature Matching Algorithm

$$\boxed{\forall P.\forall R \vdash \mathcal{F} \succeq \mathcal{F}' \ \downarrow \ \_}$$

$$\frac{\forall P \cup R \cup M.\forall\emptyset \vdash \mathcal{M}_M \succeq \mathcal{M}_N \ \downarrow \ \varphi' \quad \forall P.\forall R \cup M \vdash \varphi'\left(\mathcal{M}'_N\right) \succeq \mathcal{M}'_M \ \downarrow \ \varphi}{\forall P.\forall R \vdash \forall N.\mathcal{M}_N \to \mathcal{M}'_N \succeq \forall M.\mathcal{M}_M \to \mathcal{M}'_M \ \downarrow \ \varphi}$$

- φ is a realisation for the functors argument-result dependencies.
- contra-variance: roles of modules reversed in left premise.
- computation of φ is delayed while computing the realisation φ' of N.
- Parameters M are added to R, before computing φ: result types may depend on M.

Summary & Conclusions

Moving from First-Order to Higher-Order Modules:

- **No need to abandon the Definition:**
  - a slight change in syntax.
  - An easy generalisation of semantic objects.
- **No dependent types:**
  - Higher-order variables capture higher-order type dependencies.
  - Need higher-order realisations.
- **Flexible notion of functor enrichment:**
  - polymorphic generalisation
  - contravariant enrichment
  - enrichment and matching intertwined.
- **Retain a notion of generativity:**
  - adopt **applicative functors**.
- **Decidable type inference:**
  - There is a simple algorithm for signature matching.