

Applicative functors and fully transparent higher-order modules.

(Xavier Leroy, POPL 1995)

John S. Bucy

November 2002

# Applicative Functors

Leroy argues that the generativity enforced by SML is too strict.

Proposes *applicative* functors which are just like normal functors except that they “map provably equal arguments to compatible abstract types.”

Also claims that applicative functors allow you to give fully syntactic signatures for higher-order functors.

Leroy’s Solution: Allow functor applications in type paths.

## Local application example

```
signature ORD = sig ... end
functor Set(Elt : ORD) = struct ... end
```

```
signature DICT = sig ... end
functor Dict(Key : ORD) : DICT = struct ... end
```

Let's say we want to add a “domain” operation to Dict that returns the set of keys of a dictionary.

```
signature DICT = sig
  ...
  structure KeySet : SET sharing KeySet.elm = key
  val domain : 'a dict -> KeySet.set
```

end

```
functor Dict(Key : ORD) : DICT = struct
  ...
  structure KeySet = Set(Key)
  fun domain (d : 'a dict) : KeySet.set = ...
end
```

Say we also had some other code lying around that also had Set(Key) as a substructure:

```
functor PrioQueue(Elt : ORD) : PRIOQUEUE =
  ...
  structure EltSet = Set(Elt)
  fun contents : (q : queue) : EltSet.set = ...
end
```

So if we applied `PrioQueue` and `Dict` to the same ORD structure, the result type of the `domain` and `contents` functions are incompatible even though they are both the same type.

## The SML Solution

Lift the set up to a second functor argument.

```
functor Dict(structure Key : ORD
             structure Set : SET
             sharing type Key.t = Set.elt) = ...
```

```
functor PrioQueue(structure Elt : ORD
                  structure Set : SET
                  sharing Set.elt = Elt.t) = ...
```

Leroy argues against this since we're making a conservative extension of the old Dict. We now have to change every use of Dict and PrioQueue. (are there really all that many?)

## But its easy with applicative functors

```
functor Dict(Key : ORD) : sig
  ...
  val domain : 'a dict -> Set(Key).set
end
```

```
functor PrioQueue(Elt : ORD) : sig
  ...
  val contents : queue -> Set(Elt).set
end
```

So the result types are now compatible provided Dict and PrioQueue are applied to the same structure.

## Higher-order full transparency example

```
signature S = sig type t end
functor Apply(functor F(X:S):S structure A:S) = F(A)
```

Without applicative functors, the most general signature we can give the result of `Apply` is `S`.

This fails to propagate type equivalences.

With applicative functors, we can give it

```
functor Apply(functor F(X:S):S structure A:S) : sig
  type t = F(A).t
end
```

We run into the lambda-lifting issue again here since we can only apply functors to paths.

```
functor ApplyProd(F(X:S):S A:S) =  
F(struct type t = A.t * A.t end)
```

We can't write down the result signature since we are still limited to applying structures to paths.

So we bind up the “product” operation in another functor beforehand.

```
functor Prod (A:S) = struct type t = A.t * A.t end  
functor ApplyProd(F, A) = F(Prod(A))
```

So the result signature of ApplyProd is now

```
sig type t = F(Prod(A)).t end
```

## Is this safe?

1st-class applicative functors are unsound in the presence of effects:

```
if moonFull() then int else bool
```

Leroy obliquely mentions that value components of structures may depend on state. He also mentions that one may re-ascribe a structure to force the creation of new types “for extra safety.”

But safety from what??

## Weird example

```
signature S = sig
  type t
  val x : t
  val f : t -> t
  val g : (t * t -> bool)
end
```

```
structure Weird :> S = struct
  type t = int
  val x = if moonFull() then 1 else 2
  fun f(x) = x+2
  val g(x,y) = ((3*x + 2*y) div (x - y + 1)) = 7
end
```

## Weird Example

Consider:

```
structure W1 = ID(Weird) (* moon full now *)  
structure W2 = ID(Weird) (* moon no longer full now *)
```

If  $F$  is applicative,  $W1.t$  is compatible with  $W2.t$  so  $W1.g(W1.x, W2.x)$  is well-typed.

But this yields “uncaught exception divide by zero”!

## Whose fault is this?

Not necessarily Weird's author's fault. He had in his head at the time he wrote `g` the invariant:  $x$  and  $y$  are either both even or both odd since the only way the user has to create new ones maps even to even and odd to odd.

At least in this case, we can get the type system to help us some:

## Something of a fix

```
structure Weird' :> S = struct
  datatype t = Even of int | Odd of int
  val x = if moonFull() then Odd(1) else Even(2)

  ...

  local fun realG(x,y) = ((3*x + 2*y) div (x - y + 1)) = 7 in
    val g(Even(x),Even(y)) = realG(x,y)
      | g(Odd(x),Odd(y)) = realG(x,y)
      | g(_) = raise SomeOtherException
  end
end
```

## Something of a fix

This is really klutzy and the client code *still* gets a runtime exception.

The client could use ascription to make sure that W1.t and W2.t aren't compatible but client shouldn't have to care.

What if the implementor *doesn't know* that his code has this weird property?

This situation seems to occur quite often in real code.

So ... exploiting the applicativity in client code may yield unexpected runtime behavior.

# The Applicative Functor Calculus

Very simple extension; operational semantics pretty unremarkable.

Seems to get labels vs names right; an identifier  $x_i$  has a name part  $x$  and a stamp part  $i$  to allow  $\alpha$ -conversion inside the body of a structure.

Extend paths with functor application:  $p ::= x_i | p.x | p_1(p_2)$

## Typing Judgements

$\Gamma \vdash m : M$	module $m$ has module type $M$
$\Gamma \vdash M_1 \leq M_2$	module type $M_1$ a subtype of $M_2$
$\Gamma \vdash M \text{ modtype}$	well-formedness of $M$
$\Gamma \vdash P \text{ ok}$	program $P$ is well-typed

## Typing Rules:

- Module Expressions
- Module subtyping
- Type equivalence
- Well-formedness of module types (Leroy omits)

# Typing Rules

Substructure access (2)

$$\frac{\Gamma \vdash p : (\text{sig } S_1; \text{module } x_i : M; S_2 \text{ end})}{\Gamma \vdash p.x : M\{n_i \leftarrow p.n \mid n_i \in BV(S_1)\}}$$

Functor Application (5)

$$\frac{\Gamma \vdash m_1 : \text{functor}(x_i : M)M' \quad \Gamma \vdash m_2 : M}{\Gamma \vdash m_1(m_2) : (M'\{x_i \leftarrow m_2\})}$$

## Selfification (8)

$$\frac{\Gamma \vdash p : M}{\Gamma \vdash p : M/p}$$

Where

$$(\text{sig } S \text{ end } )/p = \text{sig } S/p \text{ end}$$

$$(\text{functor } (x_i : M_1) : M_2)/p = \text{functor } (x_i : M_1)(M_2/p(x_i))$$

$$(\text{type } t_i)/p = \text{type } t_i = p.t$$

$$(\text{type } t_i = \tau)/p = \text{type } t_i = p.t$$

$$(\text{module } x_i : M)/p = \text{module } x_i : M/p.x$$

## Type Equivalence (22,23)

$$\Gamma_1; \text{ type } t_i = \tau; \Gamma_2 \vdash t_i \approx \tau$$
$$\frac{\Gamma \vdash p : \text{sig } S_1; \text{ type } t_i = \tau; S_2 \text{ end}}{\Gamma \vdash p.t \approx \tau \{n_i \leftarrow p.n \mid n_i \in BV(S_1)\}}$$

# Representation Independence

Basic idea: abstraction is enforced if client code evaluates the same when different implementations of the abstraction are “plugged in” to it; in other words, the client code *can't differentiate between the implementations*.

Leroy gives a straightforward denotational semantics via  $[[\cdot]]_\rho$ .

# Notation

Tricky; sometimes, Leroy uses  $\Gamma$  for type environments, elsewhere, for interpretation environments.

# Interpretations

The interpretation of a structure maps type names to base relations and module names to module interpretations.

The interpretation of a functor is a function from module interpretations to module interpretations.

An interpretation environment maps type identifiers to base relations and module identifiers to module interpretations.

# Logical Relations

Start with a family  $(R_{\Gamma}^{\tau})$  of logical relations over values of the base language.

e.g.  $(b_1, b_2) \in R_{\Gamma}^{\text{int}}$  iff  $b_1$  and  $b_2$  are equal integers.

$(b_1, b_2) \in R_{\Gamma}^{\tau \rightarrow \sigma}$  iff for every  $(a_1, a_2)$  related at  $\tau$ ,  $b_1(a_1)$  and  $b_2(a_2)$  are related at  $\sigma$ .

# Logical Relations

Build up a relation between values inductively on module types.

$$\Gamma \models v \approx v' : M \Rightarrow T$$

In English, “in the type interpretation  $\Gamma$ , values  $v$  and  $v'$  are equivalent at type  $M$  yielding interpretation  $T$ .”

$\Gamma$  assigns meanings to the type identifiers and type paths in  $M$ .

$T$  records the relations we used to prove  $v$  and  $v'$  equivalent.

# Logical Relations

For evaluation environments:

$$\Gamma \models \rho \approx \rho' : E \Rightarrow \Gamma'$$

$\rho, \rho'$  evaluation environments,  $\Gamma'$  extends  $\Gamma$  with interpretations of identifiers in  $E$ .

## Logical Relations: Main Lemma

If  $E \vdash m : M$  and  $\emptyset \models \rho \approx \rho' : E \Rightarrow \Gamma$  then there's some set of relations proving that the results of evaluating  $m$  in the environments  $\rho$  and  $\rho'$  are equivalent when viewed at  $M$ .

i.e.  $\exists T$  such that

$$\Gamma \models [[m]]_{\rho} \approx [[m]]_{\rho'} : M \Rightarrow T$$

## A Corollary

Let  $m_1$  and  $m_2$  be closed module expressions with type  $M$  in the empty type environment. If  $\emptyset \models [[m_1]]_\emptyset \approx [[m_2]]_\emptyset : M \Rightarrow T$  for some  $T$  then

$$\forall C [].x_i : M \vdash C[x_i] \text{ok} \rightarrow [[C[m_1]]] = [[C[m_2]]]$$

## The Main Result

Would like to “take advantage of more typing hypotheses that could be derived about a particular implementation,” so lift requirement that the “minimal” environment  $x_i : M$  type  $C[x_i]$ .

Restated: if  $M$  is principal for  $m_1$  and  $m_2$  in the empty environment, then for all  $C[]$   $C[m_1]$  well-typed iff  $C[m_2]$  is and if so,  $[[C[m_1]]] = [[C[m_2]]]$ .

## Full transparency for higher-order functors

Idea: take MacQueen and Tofte's calculus and show that all of its "correct" programs are well-typed in the applicative functors calculus.

Says that this is too hard, so instead, we'll encode the "manifest sums" calculus (like XML or "manifest types") into it instead.

"The existence of a type-preserving encoding into the calculus with applicative functors is therefore a strong hint that the latter ensures full transparency."

# The Manifest Sums Calculus

$$m ::= x_i | \lambda x_i : M.m | m_1(m_2) | \iota_v(e) | \iota_t(t) | \langle x_i = m_1, m_2 \rangle | \pi_1(m) | \pi_2(m)$$

$\iota_v$  injection for terms (values),  $\iota_t$  for types

Form structures by pairing injections of simple terms and types.

$$M ::= V(\tau) | \text{TYPE} | \text{EQ}(\tau) | \Sigma x_i : M_1.M_2 | \Pi x_i : M_1.M_2$$

Functors are  $\lambda$ -abstractions with  $\Pi$  types.

$$e ::= \dots | \pi_v(m)$$

$$\tau ::= \dots | \pi_t(m)$$

# Path Normalization

We have make terms conform to applicative functor calculus's stricter syntactic requirements, i.e. restriction of functor application to paths.

Paths in manifest sums calculus are  $p ::= x_i | \pi_1(p) | \pi_2(p) | p_1(p_2)$

Add let bindings for terms and types.

## Rewrite Rules: Bind Projection Arguments

“The first group of rewrite rules introduce names for arguments to projections that are not paths.”

Terms:  $\iota_v(e[\pi_v(m_c)]) \rightarrow \mathbf{let} \ x_i = m_c \mathbf{in} \ \iota_v(e[\pi_v(x_i)])$

Types:  $V(\tau[\pi_t(m_c)]) \rightarrow \mathbf{Let} \ x_i = m \mathbf{in} \ V(\tau[\pi_t(x_i)])$

## Rewrite Rules: Lift Let

“The second group of rules lift let bindings upwards until they hit a pair operator.”

Terms:  $\text{let } \sigma \text{ in let } \sigma' \text{ in } m \rightarrow \text{let } \sigma; \sigma' \text{ in } m$

$\lambda x_i : (\text{Let } \sigma \text{ in } M) \rightarrow \text{let } \sigma \text{ in } \lambda x_i : M.m$

## Rewrite Result

Claim: after rewriting, the paths are all of the form  $p ::= x_i | \pi_1(p) | \pi_2(p) | p_1(p_2)$

Also, we've eliminated all of the Let bindings.

Only applications are to paths.

let only appears at top-level or in arguments to a pair.

# The Encoding

$$\llbracket x_i \rrbracket = x_i$$

$$\llbracket \pi_1(p) \rrbracket = \llbracket p \rrbracket.\text{fst}$$

$$\llbracket \pi_2(p) \rrbracket = \llbracket p \rrbracket.\text{snd}$$

$$\llbracket \lambda x_i : M.m \rrbracket = \text{functor } (x_i : \llbracket M \rrbracket) \llbracket m \rrbracket$$

$$\llbracket m(p) \rrbracket = \llbracket m \rrbracket(\llbracket p \rrbracket)$$

$$\llbracket \iota_v(e) \rrbracket = \text{struct val } v_i = \llbracket e \rrbracket \text{end}$$

$$\llbracket \iota_t(\tau) \rrbracket = \text{struct type } t_i = \llbracket \tau \rrbracket \text{end}$$

## The Encoding: Dependent Pairs

$\llbracket \langle x_i = (\text{let } \sigma_1 \text{ in } m_1), \text{let } \sigma_2 \text{ in } m_2 \rangle \rrbracket =$

`struct  $\llbracket \sigma_1 \rrbracket$ ;`

`module fst  $j = \llbracket m_1 \rrbracket$ ;`

`$\llbracket \sigma_2 \rrbracket \{x_i \leftarrow \text{fst } j\}$ ;`

`module snd  $k = \llbracket m_2 \rrbracket \{x_i \leftarrow \text{fst } j\}$`

`end`

## Encoding Substitutions and Types

Substitutions (introduced by let-bindings) encoded as a sequence of module bindings:  $\llbracket x_1 = m_1 \rrbracket$  is `module  $x_1 = \llbracket m_1 \rrbracket$  end`.

$$\llbracket V(\tau) \rrbracket = \text{sig val } v_i : \llbracket \tau \rrbracket \text{ end}$$
$$\llbracket TYPE \rrbracket = \text{sig type } t \text{ end}$$
$$\llbracket EQ(\tau) \rrbracket = \text{sig type } t_i = \llbracket \tau \rrbracket \text{ end}$$
$$\llbracket \pi_v(p) \rrbracket = \llbracket p \rrbracket.v$$
$$\llbracket \pi_t(p) \rrbracket = \llbracket p \rrbracket.t$$

## Encoding $\Sigma$ and $\Pi$

$\llbracket \Sigma x_i : M_1.M_2 \rrbracket = \text{sig } \text{module } fst_j : \llbracket M_1 \rrbracket; \text{ module } snd_k :$   
 $\llbracket M_2 \rrbracket \{x_i \leftarrow fst_j\} \text{end}$

$\llbracket \Pi x_i : M_1.M_2 \rrbracket = \text{functor } (x_i : \llbracket M_1 \rrbracket) \llbracket M_2 \rrbracket$

# Type Preservation

Leroy asserts that if  $E \vdash m : M$  in the manifest sums calculus, then  $\llbracket E \rrbracket \vdash \llbracket m \rrbracket : \llbracket M \rrbracket$  in the applicative functor calculus.

## Conclusions

Applicative functors seem handy but Leroy's implementation has the serious problem that client code can accidentally violate invariants of structures which have stateful value components.

I think this violates abstraction at least in spirit.

Does the representation independence proof work if the base language has references and exceptions?

Does the manifest sums calculus really capture everything in e.g. MacQueen and Tofte's higher-order functor calculus?