F-ing Applicative Functors

Andreas Rossberg, Google
Claudio Russo, MSR
Derek Dreyer, MPI-SWS

ML Workshop, Copenhagen 2012/09/13

The Functor Schism

The Functor Schism

- SML: “generative” functors

  ⇒ return “fresh” abstract types with each application

The Functor Schism

- SML: “generative” functors

  ⇒ return “fresh” abstract types with each application

- OCaml: “applicative” functors

  ⇒ return same abstract types with each application

  (to the “same” argument)

The Functor Schism

- SML: “generative” functors

  ⇒ return “fresh” abstract types with each application

- OCaml: “applicative” functors

  ⇒ return same abstract types with each application

  (to the “same” argument)

Example: Set

Example: Set

```
signature SET =
{
    type elem
    type set
    val empty : set
    val add : elem → set → set
    val member : elem → set → bool
}
```

Example: Set

```
signature SET =
{
    type elem
    type set
    val empty : set
    val add : elem → set → set
    val member : elem → set → bool
}

signature ORD =
{
    type t
    val leq : t → t → bool
}
```

Example: Set

```
signature SET =
{
    type elem
    type set
    val empty : set
    val add : elem → set → set
    val member : elem → set → bool
}

signature ORD =
{
    type t
    val leq : t → t → bool
}

module Set (Elem : ORD) :> (SET where type elem = Elem.t) =
    { ... }
```

Example: Set, generative

module A = Set Int

A.member 3 (A.add 2 A.empty)

Example: Set, generative

module A = Set Int

module B = Set Int

A.member 3 (B.add 2 B.empty)

Example: Set, generative

```
module A = Set Int

module B = Set Int

A.member 3 (B.add 2 B.empty)      (* ill-typed, A.set ≠ B.set *)
```

Example: Set, applicative

```
module A = Set Int

module B = Set Int

A.member 3 (B.add 2 B.empty)
                    (* well-typed, A.set = Set(Int).set = B.set *)
```

The story, so far

- Leroy [POPL 1995] – OCaml
- Russo [Thesis 1998 / ENTCS 2003] – Moscow ML
- Shao [ICFP 1999]
- Dreyer & Crary & Harper [POPL 2003]

Plan

- Issues
- Proposal
- Formalisation

Why applicative functors?

Why applicative functors?

- Literature: motivated by higher-order functors

Why applicative functors?

- Literature: motivated by higher-order functors
- Practice: compilation units importing functors are higher-order functors in disguise

Example: Map

```
signature MAP =
{
    type key
    type map α
    val empty : map α
    val add : key → α → map α → map α
    val lookup : key → map α → option α
}
```

```
module Map : (Key : ORD) → MAP with type key = Key.t
```

Example: Map

```
signature MAP =
{
    type key
    type map α
    val empty : map α
    val add : key → α → map α → map α
    val lookup : key → map α → option α
    val domain : map α → ?
}
```

```
module Map : (Key : ORD) → MAP with type key = Key.t
```

With generative Set, variant 1

```
signature MAP =
{
    type key
    type map α
    module Set : SET with type elem = key
    val empty : map α
    val add : key → α → map α → map α
    val lookup : key → map α → option α
    val domain : map α → Set.set
}

module Map : (Key : ORD) → MAP with type key = Key.t
```

With generative Set, variant 2

```
signature MAP =
{
    type key
    type map α
    type set
    val empty : map α
    val add : key → α → map α → map α
    val lookup : key → map α → option α
    val domain : map α → set
}

module Map : (Key : ORD) → (Set : SET with type elem = Key.t) →
    MAP with type key = Key.t with type set = Set.t
```

With applicative Set

```
signature MAP =
{
    module Key : ORD
    type map α
    val empty : map α
    val add : Key.t → α → map α → map α
    val lookup : Key.t → map α → option α
    val domain : map α → Set(Key).set
}
```

```
module Map : (Key : ORD) → MAP with type key = Key.t
```

Why applicative functors?

- Two independent units can use the same generic data type without any need for cooperation
- A third unit using both is still able to exchange sets between them seamlessly (a.k.a. diamond import)

Observation 0

Applicative functors are useful for modularity.

Example: First-class modules

signature S = { **type** t; **val** v : t; **val** f : t → t }

val p_1 = **pack** { **type** t = int; **val** v = 6; **val** f = negate } : S
val p_2 = **pack** { **type** t = string; **val** v = “uh”; **val** f = id } : S

Example: First-class modules

signature S = { **type** t; **val** v : t; **val** f : t → t }

val p_1 = **pack** { **type** t = int; **val** v = 6; **val** f = negate } : S
val p_2 = **pack** { **type** t = string; **val** v = “uh”; **val** f = id } : S

val p = ref p_1
module F {} = **unpack** !p : S

Example: First-class modules

signature S = { **type** t; **val** v : t; **val** f : t → t }

val p_1 = **pack** { **type** t = int; **val** v = 6; **val** f = negate } : S
val p_2 = **pack** { **type** t = string; **val** v = “uh”; **val** f = id } : S

val p = ref p_1
module F {} = **unpack** !p : S

module M_1 = F {}
p := p_2
module M_2 = F {}

Example: First-class modules

signature S = { **type** t; **val** v : t; **val** f : t → t }

val p_1 = **pack** { **type** t = int; **val** v = 6; **val** f = negate } : S
val p_2 = **pack** { **type** t = string; **val** v = “uh”; **val** f = id } : S

val p = ref p_1
module F {} = **unpack** !p : S

module M_1 = F {}
p := p_2
module M_2 = F {}
M_1.f M_2.v   *(* oops, ka-boom! *)*

Observation 1

Applicativity can break type safety (with 1st-class modules).

Example: Abstract names

```
signature NAME =
{
  type name
  val new : () → name
  val eq : name → name → bool
}

module Name {} :> NAME =
{
  type name = int
  val count = ref 0
  val new () = ++count ; !count
  val eq = Int.eq
}
```

Example: Abstract names

module A = Name {}
module B = Name {}

val a = A.new ()
val b = B.new ()

Example: Abstract names

module A = Name {}
module B = Name {}

val a = A.new ()
val b = B.new ()

a = b

Example: Abstract names

```
module A = Name {}
module B = Name {}

val a = A.new ()
val b = B.new ()

a = b      (* oops, true! *)
```

Observation 2

Applicativity can break abstraction safety (of impure functors).

Example: Set ordering

```
module A = Set {type t = int; val leq = Int.leq}
module B = Set {type t = int; val leq = Int.geq}
```

Example: Set ordering

```
module A = Set {type t = int; val leq = Int.leq}
module B = Set {type t = int; val leq = Int.geq}

val s1 = A.add 2 A.empty
val s2 = B.add 3 s1
```

Example: Set ordering

module A = Set {**type** t = int; **val** leq = Int.leq}
module B = Set {**type** t = int; **val** leq = Int.geq}

val s_1 = A.add 2 A.empty
val s_2 = B.add 3 s_1

A.member 2 s_2

Example: Set ordering

```
module A = Set {type t = int; val leq = Int.leq}
module B = Set {type t = int; val leq = Int.geq}

val s1 = A.add 2 A.empty
val s2 = B.add 3 s1

A.member 2 s2      (* oops, false! *)
```

Observation 3

Applicativity can break abstraction safety (of pure functors).

Example: Set ordering, in Ocaml

module A = Set {**type** t = int; **val** leq = Int.leq}
module B = Set {**type** t = int; **val** leq = Int.geq}

Example: Set ordering, in Ocaml

module A = Set {**type** t = int; **val** leq = Int.leq}
module B = Set {**type** t = int; **val** leq = Int.geq}

val s_1 = A.add 2 A.empty
val s_2 = B.add 3 s_1

Example: Set ordering, in Ocaml

module A = Set {**type** t = int; **val** leq = Int.leq}
module B = Set {**type** t = int; **val** leq = Int.geq}

val s_1 = A.add 2 A.empty
val s_2 = B.add 3 s_1
(type error, A.set ≠ B.set, because Set({...}).set not a path *)*

Set ordering in Ocaml, take 2

Set ordering in Ocaml, take 2

```
module F (X : {}) =
{
    type t = int
    val leq = if isFullMoon () then Int.leq else Int.geq
}
```

Set ordering in Ocaml, take 2

```
module F (X : {}) =      (* returns one of the previous modules! *)
{
    type t = int
    val leq = if isFullMoon () then Int.leq else Int.geq
}
```

Set ordering in Ocaml, take 2

```
module F (X : {}) =      (* returns one of the previous modules! *)
{
    type t = int
    val leq = if isFullMoon () then Int.leq else Int.geq
}

module Unit = {}
module A = Set (F Unit)
module B = Set (F Unit)
```

Set ordering in Ocaml, take 2

```
module F (X : {}) =      (* returns one of the previous modules! *)
{
    type t = int
    val leq = if isFullMoon () then Int.leq else Int.geq
}

module Unit = {}
module A = Set (F Unit)
module B = Set (F Unit)

val s1 = A.add 2 A.empty
val s2 = B.add 3 s1

A.member 2 s2
```

Set ordering in Ocaml, take 2

```
module F (X : {}) =       (* returns one of the previous modules! *)
{
    type t = int
    val leq = if isFullMoon () then Int.leq else Int.geq
}

module Unit = {}
module A = Set (F Unit)
module B = Set (F Unit)

val s1 = A.add 2 A.empty
val s2 = B.add 3 s1

A.member 2 s2      (* oops, false!  A.set = Set(F Unit).set = B.set *)
```

Observation 4

Impure applications in paths breaks abstraction safety (even of pure functors).

Type Equivalence in Ocaml

module A = Set {**type** t = int; **val** leq = Int.leq}
module B = Set {**type** t = int; **val** leq = Int.leq}

val s_1 = A.add 2 A.empty
val s_2 = B.add 3 s_1

A.member 2 s_2
(type error, A.set ≠ B.set, because Set({...}).set not a path *)*

Type Equivalence in Ocaml (2)

module IntOrd = {**type** t = int; **val** leq = Int.leq}
module IntOrd' = IntOrd
module A = Set IntOrd
module B = Set IntOrd'

val s_1 = A.add 2 A.empty
val s_2 = B.add 3 s_1

A.member 2 s_2
(type error, A.set = Set(IntOrd).set ≠ Set(IntOrd').set = B.set *)*

Observation 5

Syntactic path equivalence is overly restrictive.

Hm...

Overcoming the Schism

Overcoming the Schism

- Support both applicative and generative functors

Overcoming the Schism

- Support both applicative and generative functors
- A functor is applicative if and only if it is pure

Overcoming the Schism

- Support both applicative and generative functors
- A functor is applicative if and only if it is pure

    ⇒ type system tracks purity

Overcoming the Schism

- Support both applicative and generative functors
- A functor is applicative if and only if it is pure

  ⇒ type system tracks purity

- Two modules are equivalent if and only if they define equivalent types *and* values

Overcoming the Schism

- Support both applicative and generative functors
- A functor is applicative if and only if it is pure

  ⇒ type system tracks purity

- Two modules are equivalent if and only if they define equivalent types *and* values

  ⇒ type system tracks value identity
  (while avoiding dependent types)

Purity

Purity

- Only one form of functor expression, deemed pure iff:
  - it does not unpack a first-class module
  - it does not apply an impure functor
  - all value bindings are “non-expansive” (value restriction)

Purity

- Only one form of functor expression, deemed pure iff:
  - it does not unpack a first-class module
  - it does not apply an impure functor
  - all value bindings are “non-expansive” (value restriction)
- Two forms of functor type
  - impure: $(X : S_1) \rightarrow S_2$
  - pure: $(X : S_1) \Rightarrow S_2$

Abstract Values

Abstract Values

- Every value binding is identified by an abstract value

Abstract Values

- Every value binding is identified by an abstract value
    - Mere renamings retain identity (e.g., **val** x = A.y)

Abstract Values

- Every value binding is identified by an abstract value
  - Mere renamings retain identity (e.g., **val** x = A.y)
  - Other bindings define fresh abstract value

Abstract Values

- Every value binding is identified by an abstract value
  - Mere renamings retain identity (e.g., **val** x = A.y)
  - Other bindings define fresh abstract value
  - Specifications (in signatures) declare abstract values

Abstract Values

- Every value binding is identified by an abstract value
  - Mere renamings retain identity (e.g., **val** x = A.y)
  - Other bindings define fresh abstract value
  - Specifications (in signatures) declare abstract values
- Formally, abstract values are phantom type variables, quantified and matched in same manner as abstract types

Abstract Values

- Every value binding is identified by an abstract value
  - Mere renamings retain identity (e.g., **val** x = A.y)
  - Other bindings define fresh abstract value
  - Specifications (in signatures) declare abstract values
- Formally, abstract values are phantom type variables, quantified and matched in same manner as abstract types
- Refinement of SML90's structure sharing

Module Syntax

$$
\begin{array}{llll}
\text{Modules} & M & ::= & X \\
& & & \{B\} \\
& & & M.X \\
& & & \textbf{fun}\ X{:}S \Rightarrow M \\
& & & X\ X \\
& & & X{:>}S
\end{array}
\qquad
\begin{array}{llll}
\text{Signatures} & S & ::= & X \\
& & & \{D\} \\
& & & M.X \\
& & & (X{:}S) \to S \\
& & & (X{:}S) \Rightarrow S \\
& & & S\ \textbf{where type}\ \overline{X}{=}T
\end{array}
$$

$$
\begin{array}{llll}
\text{Bindings} & B & ::= & \textbf{val}\ X{=}E \\
& & & \textbf{type}\ X{=}T \\
& & & \textbf{module}\ X{=}M \\
& & & \textbf{signature}\ X{=}S \\
& & & \textbf{include}\ M \\
& & & B;B \\
& & & \epsilon
\end{array}
\qquad
\begin{array}{llll}
\text{Declarations} & D & ::= & \textbf{val}\ X{:}T \\
& & & \textbf{type}\ X{=}T \\
& & & \textbf{type}\ X{:}K \\
& & & \textbf{module}\ X{:}S \\
& & & \textbf{signature}\ X{=}S \\
& & & \textbf{include}\ S \\
& & & D;D \\
& & & \epsilon
\end{array}
$$

F-ing Formalisation

F-ing Elaboration, recap

Signatures $\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma$

Modules $\Gamma \vdash M : \exists \overline{\alpha}.\Sigma \rightsquigarrow e$

F-ing Elaboration, recap

Signatures	$\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma$	$\Rightarrow$	$\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$
Modules	$\Gamma \vdash M : \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$\Rightarrow$	$\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$

Semantic Signatures, recap

$$
\begin{array}{rcll}
\Sigma & ::= & [\tau] & \text{(term)} \\
& | & [= \tau{:}\kappa] & \text{(type)} \\
& | & \{\overline{l : \Sigma}\} & \text{(structure)} \\
& | & \forall \overline{\alpha}_1.\Sigma_1 \to \exists \overline{\alpha}_2.\Sigma_2 & \text{(functor)}
\end{array}
$$

Example: Set Signature

Set : (Elem : ORD) → (SET **where type** elem = Elem.t)

$$
\begin{array}{l}
\forall\alpha.\{ \\
\quad \text{t} : [= \alpha : \Omega], \\
\quad \text{leq} : [\alpha \to \alpha \to bool] \\
\} \to \\
\quad \exists\beta.\ \{ \\
\quad\quad \text{elem} : [= \alpha : \Omega], \\
\quad\quad \text{set} : [= \beta : \Omega], \\
\quad\quad \text{empty} : [\beta], \\
\quad\quad \text{add} : [\alpha \to \beta \to \beta], \\
\quad\quad \text{member} : [\alpha \to \beta \to bool] \\
\quad \}
\end{array}
$$

Elaboration, revised

Signatures	$\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma$	
Modules	$\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$(\varphi ::= \mathtt{P} \mid \mathtt{I})$

Semantic Signatures, revised

$$
\begin{array}{lcll}
\pi & ::= & \alpha\ \overline{\tau} & \text{(path)} \\
\Sigma & ::= & [= \pi{:}\tau] & \text{(term)} \\
 & | & [= \tau{:}\kappa] & \text{(type)} \\
 & | & \{\overline{l : \Sigma}\} & \text{(structure)} \\
 & | & \forall\overline{\alpha}_1.\Sigma_2 \rightarrow_{\varphi} \exists\overline{\alpha}_2.\Sigma_2 & \text{(functor)}
\end{array}
$$

Semantic Signatures, revised

$$\begin{array}{rcll}
\pi & ::= & \alpha\ \overline{\tau} & \text{(path)} \\
\Sigma & ::= & [= \pi{:}\tau] & \text{(term)} \\
& | & [= \tau{:}\kappa] & \text{(type)} \\
& | & \{\overline{l : \Sigma}\} & \text{(structure)} \\
& | & \forall\overline{\alpha}_1.\Sigma_2 \rightarrow_\varphi \exists\overline{\alpha}_2.\Sigma_2 & \text{(functor)}
\end{array}$$

Impure functor: $\forall\overline{\alpha}_1.\Sigma_1 \rightarrow_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2$

Pure functor: $\exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\Sigma_1 \rightarrow_{\mathtt{P}} \Sigma_2$

Functor Signatures

$$\boxed{\Gamma \vdash S \rightsquigarrow \exists \overline{\alpha}.\Sigma}$$

Functor Signatures

$$\boxed{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2}{\Gamma \vdash (X{:}S_1) \rightarrow S_2 \rightsquigarrow \forall\overline{\alpha}_1.\,\Sigma_1 \rightarrow \exists\overline{\alpha}_2.\Sigma_2}$$

Functor Signatures

$$\boxed{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2}{\Gamma \vdash (X{:}S_1) \to S_2 \rightsquigarrow \forall\overline{\alpha}_1.\,\Sigma_1 \to \exists\overline{\alpha}_2.\Sigma_2}$$

$$\frac{\Gamma \vdash S_1 \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash S_2 \rightsquigarrow \exists\overline{\alpha}_2.\Sigma_2}{\Gamma \vdash (X{:}S_1) \Rightarrow S_2 \rightsquigarrow \exists\overline{\alpha}_2'.\forall\overline{\alpha}_1.\,\Sigma_1 \to \Sigma_2[\overline{\alpha_2'\ \overline{\alpha}_1/\alpha_2}]}$$

$$\overline{\alpha}_1 : \overline{\kappa}_1$$
$$\overline{\alpha}_2 : \overline{\kappa}_2$$
$$\overline{\alpha}_2' : \overline{\overline{\kappa}_1 \to \kappa_2}$$

Example: Set Signature

Set : (Elem : ORD) ⇒ (SET **where type** elem = Elem.t)

$$
\begin{array}{l}
\exists \beta_{\Omega \to \Omega}.\ \forall \alpha. \{ \\
\quad \mathsf{t} : [= \alpha : \Omega], \\
\quad \mathsf{leq} : [\alpha \to \alpha \to bool] \\
\} \to \\
\quad \{ \\
\quad\quad \mathsf{elem} : [= \alpha : \Omega], \\
\quad\quad \mathsf{set} : [= \beta\, \alpha : \Omega], \\
\quad\quad \mathsf{empty} : [\beta\, \alpha], \\
\quad\quad \mathsf{add} : [\alpha \to \beta\, \alpha \to \beta\, \alpha], \\
\quad\quad \mathsf{member} : [\alpha \to \beta\, \alpha \to bool] \\
\quad \}
\end{array}
$$

Example: Set Signature

Set : (Elem : ORD) ⇒ (SET **where type** elem = Elem.t)

$$
\begin{array}{l}
\exists \beta\, \beta_1 \beta_2 \beta_3.\ \forall \alpha\, \alpha_1 . \{ \\
\quad \mathsf{t} : [= \alpha : \Omega], \\
\quad \mathsf{leq} : [= \alpha_1 : \alpha \to \alpha \to \mathit{bool}] \\
\} \to \\
\quad \{ \\
\qquad \mathsf{elem} : [= \alpha : \Omega], \\
\qquad \mathsf{set} : [= \beta\, \alpha\, \alpha_1 : \Omega], \\
\qquad \mathsf{empty} : [= \beta_1 : \beta\, \alpha\, \alpha_1], \\
\qquad \mathsf{add} : [= \beta_2 : \alpha \to \beta\, \alpha\, \alpha_1 \to \beta\, \alpha\, \alpha_1], \\
\qquad \mathsf{member} : [= \beta_3 : \alpha \to \beta\, \alpha\, \alpha_1 \to \mathit{bool}] \\
\quad \}
\end{array}
$$

Functor Expressions

$$\boxed{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}$$

Functor Expressions

$$\boxed{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash M :_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M :_{\mathtt{P}} \forall\overline{\alpha}_1.\,\Sigma_1 \rightarrow \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow \lambda\overline{\alpha}_1.\lambda X{:}\Sigma_1.e}$$

Functor Expressions

$$\boxed{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash M :_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M :_{\mathtt{P}} \forall\overline{\alpha}_1.\,\Sigma_1 \to \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow \lambda\overline{\alpha}_1.\lambda X{:}\Sigma_1.e}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash M :_{\mathtt{P}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M :_{\mathtt{P}} \exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\,\Sigma_1 \to \Sigma_2 \rightsquigarrow ???}$$

Elaboration Invariant, revised

Signatures	$\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma$	$\Rightarrow$	$\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$
Modules	$\Gamma \vdash M :_{\mathrm{I}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$\Rightarrow$	$\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$

Elaboration Invariant, revised

Signatures	$\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma$	$\Rightarrow$	$\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$
Modules	$\Gamma \vdash M :_{\mathtt{I}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$\Rightarrow$	$\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$
	$\Gamma \vdash M :_{\mathtt{P}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$		

Elaboration Invariant, revised

Signatures	$\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma$	$\Rightarrow$	$\Gamma \vdash \exists\overline{\alpha}.\Sigma : \Omega$
Modules	$\Gamma \vdash M :_{\mathtt{I}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$\Rightarrow$	$\Gamma \vdash e : \exists\overline{\alpha}.\Sigma$
	$\Gamma \vdash M :_{\mathtt{P}} \exists\overline{\alpha}.\Sigma \rightsquigarrow e$	$\Rightarrow$	$\cdot \vdash e : \exists\overline{\alpha}.\forall\Gamma.\Sigma$

Functor Expressions

$$\boxed{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash M :_{\mathtt{I}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M :_{\mathtt{P}} \forall\overline{\alpha}_1.\,\Sigma_1 \to \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow \lambda\Gamma.\lambda\overline{\alpha}_1.\lambda X{:}\Sigma_1.e}$$

$$\frac{\Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}_1.\Sigma_1 \qquad \Gamma, \overline{\alpha}_1, X{:}\Sigma_1 \vdash M :_{\mathtt{P}} \exists\overline{\alpha}_2.\Sigma_2 \rightsquigarrow e}{\Gamma \vdash \textbf{fun}\ X{:}S \Rightarrow M :_{\mathtt{P}} \exists\overline{\alpha}_2.\forall\overline{\alpha}_1.\,\Sigma_1 \to \Sigma_2 \rightsquigarrow e}$$

Sealing

$$\boxed{\Gamma \vdash M :_{\varphi} \exists\overline{\alpha}.\Sigma \rightsquigarrow e}$$

$$\frac{\Gamma(X) = \Sigma' \qquad \Gamma \vdash S \rightsquigarrow \exists\overline{\alpha}.\Sigma \qquad \Gamma \vdash \Sigma' \leq \exists\overline{\alpha}.\Sigma \uparrow \overline{\tau} \rightsquigarrow f}{\Gamma \vdash X :> S :_{\mathsf{P}} \exists\overline{\alpha}'.\Sigma[\overline{\alpha'\,\Gamma/\alpha}] \rightsquigarrow \mathsf{pack}\,\langle\overline{\lambda\Gamma.\tau}, \lambda\Gamma.f\,X\rangle}$$

$$\begin{array}{l} \overline{\alpha} \ : \overline{\kappa} \\ \overline{\alpha}' : \overline{\Gamma \to \kappa} \end{array}$$

Elaborating Specifications

Elaborating Specifications

$$\boxed{\Gamma \vdash D \leadsto \Xi}$$

Elaborating Specifications

$$\boxed{\Gamma \vdash D \leadsto \Xi}$$

$$\frac{\Gamma \vdash K \leadsto \kappa_\alpha}{\Gamma \vdash \textbf{type}\ X{:}K \leadsto \exists\alpha.\{X : [= \alpha : \kappa_\alpha]\}}$$

Elaborating Specifications

$$\boxed{\Gamma \vdash D \rightsquigarrow \Xi}$$

$$\frac{\Gamma \vdash K \rightsquigarrow \kappa_\alpha}{\Gamma \vdash \textbf{type } X{:}K \rightsquigarrow \exists\alpha.\{X : [= \alpha : \kappa_\alpha]\}}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type } X{=}T \rightsquigarrow \{X : [= \tau : \kappa]\}}$$

Elaborating Specifications

$$\boxed{\Gamma \vdash D \leadsto \Xi}$$

$$\frac{\Gamma \vdash K \leadsto \kappa_\alpha}{\Gamma \vdash \textbf{type } X{:}K \leadsto \exists\alpha.\{X : [= \alpha : \kappa_\alpha]\}}$$

$$\frac{\Gamma \vdash T : \kappa \leadsto \tau}{\Gamma \vdash \textbf{type } X{=}T \leadsto \{X : [= \tau : \kappa]\}}$$

$$\frac{\Gamma \vdash T : \Omega \leadsto \tau}{\Gamma \vdash \textbf{val } X{:}T \leadsto \exists\alpha.\{X : [= \alpha : \Omega]\}}$$

Elaborating Specifications

$$\boxed{\Gamma \vdash D \leadsto \Xi}$$

$$\frac{\Gamma \vdash K \leadsto \kappa_\alpha}{\Gamma \vdash \textbf{type } X{:}K \leadsto \exists\alpha.\{X : [= \alpha : \kappa_\alpha]\}}$$

$$\frac{\Gamma \vdash T : \kappa \leadsto \tau}{\Gamma \vdash \textbf{type } X{=}T \leadsto \{X : [= \tau : \kappa]\}}$$

$$\frac{\Gamma \vdash T : \Omega \leadsto \tau}{\Gamma \vdash \textbf{val } X{:}T \leadsto \exists\alpha.\{X : [= \alpha : \Omega]\}}$$

$$\frac{\Gamma \vdash P : [= \pi : \tau] \leadsto e}{\Gamma \vdash \textbf{val } X{=}P \leadsto \{X : [= \pi : \tau]\}}$$

Elaborating Bindings

Elaborating Bindings

$$\boxed{\Gamma \vdash B : \Xi \rightsquigarrow e}$$

Elaborating Bindings

$$\boxed{\Gamma \vdash B : \Xi \rightsquigarrow e}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type } X{=}T :_{\varphi} \{X : [= \tau : \kappa]\} \rightsquigarrow \{X = [\tau : \kappa]\}}$$

Elaborating Bindings

$$\boxed{\Gamma \vdash B : \Xi \leadsto e}$$

$$\frac{\Gamma \vdash T : \kappa \leadsto \tau}{\Gamma \vdash \textbf{type}\ X{=}T :_{\varphi} \{X : [= \tau : \kappa]\} \leadsto \{X = [\tau : \kappa]\}}$$

$$\frac{\Gamma \vdash E : \tau \leadsto e}{\Gamma \vdash \textbf{val}\ X{=}E :_{\mathtt{I}} \exists\alpha.\{X : [= \alpha : \tau]\} \leadsto \mathsf{pack}\ \langle\{\},\{X = [e]\}\rangle}$$

Elaborating Bindings

$$\boxed{\Gamma \vdash B : \Xi \rightsquigarrow e}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type}\ X{=}T :_{\varphi} \{X : [= \tau : \kappa]\} \rightsquigarrow \{X = [\tau : \kappa]\}}$$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e}{\Gamma \vdash \textbf{val}\ X{=}E :_{\mathtt{I}} \exists\alpha.\{X : [= \alpha : \tau]\} \rightsquigarrow \mathsf{pack}\ \langle\{\},\{X = [e]\}\rangle}$$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e \qquad E \text{ non-expansive}}{\Gamma \vdash \textbf{val}\ X{=}E :_{\mathtt{P}} \exists\alpha.\{X : [= \alpha : \tau]\} \rightsquigarrow \mathsf{pack}\ \langle\lambda\Gamma.\{\}, \lambda\Gamma.\{X = [e]\}\rangle}$$

Elaborating Bindings

$$\boxed{\Gamma \vdash B : \Xi \rightsquigarrow e}$$

$$\frac{\Gamma \vdash T : \kappa \rightsquigarrow \tau}{\Gamma \vdash \textbf{type}\ X{=}T :_{\varphi} \{X : [= \tau : \kappa]\} \rightsquigarrow \{X = [\tau : \kappa]\}}$$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e}{\Gamma \vdash \textbf{val}\ X{=}E :_{\mathtt{I}} \exists\alpha.\{X : [= \alpha : \tau]\} \rightsquigarrow \textsf{pack}\ \langle\{\},\{X = [e]\}\rangle}$$

$$\frac{\Gamma \vdash E : \tau \rightsquigarrow e \qquad E \text{ non-expansive}}{\Gamma \vdash \textbf{val}\ X{=}E :_{\mathtt{P}} \exists\alpha.\{X : [= \alpha : \tau]\} \rightsquigarrow \textsf{pack}\ \langle\lambda\Gamma.\{\}, \lambda\Gamma.\{X = [e]\}\rangle}$$

$$\frac{\Gamma \vdash P :_{\varphi} [= \pi : \tau] \rightsquigarrow e}{\Gamma \vdash \textbf{val}\ X{=}P :_{\varphi} \{X : [= \pi : \tau]\} \rightsquigarrow \{X = e\}}$$

Bonus: Sharing specifications

Bonus: Sharing specifications

- opaque & transparent value specifications
  val x : t vs. **val** x = A.y

Bonus: Sharing specifications

- opaque & transparent value specifications
  val x : t   vs.   **val** x = A.y
- opaque & transparent module specifications
  module X : S   vs.   **module** X = A.Y

Bonus: Sharing specifications

- opaque & transparent value specifications
  val x : t   vs.   **val** x = A.y
- opaque & transparent module specifications
  module X : S   vs.   **module** X = A.Y
- value & module refinements
  S **where val** X.y = z   or   S **where module** X.Y = Z

Bonus: Sharing specifications

- opaque & transparent value specifications
  val x : t vs. **val** x = A.y
- opaque & transparent module specifications
  module X : S vs. **module** X = A.Y
- value & module refinements
  S **where val** X.y = z or S **where module** X.Y = Z
- singleton signatures
  like X.Y

Conclusion

Conclusion

- Applicative functors are delicate

Conclusion

- Applicative functors are delicate
- Applicative $\Leftrightarrow$ pure, generative $\Leftrightarrow$ impure

Conclusion

- Applicative functors are delicate
- Applicative $\Leftrightarrow$ pure, generative $\Leftrightarrow$ impure
- Module equivalence = type equivalence + value equivalence

Conclusion

- Applicative functors are delicate
- Applicative ⇔ pure, generative ⇔ impure
- Module equivalence = type equivalence + value equivalence
- F-ing modules allows fairly elegant formalisation

Conclusion

- Applicative functors are delicate
- Applicative ⇔ pure, generative ⇔ impure
- Module equivalence = type equivalence + value equivalence
- F-ing modules allows fairly elegant formalisation
- Gory details in draft article:
  http://www.mpi-sws.org/~rossberg/f-ing/

Thank you!

Outtakes

Applicative Semantics for Functors is difficult!

	Leroy	Russo	Shao	Dreyer+
unrestricted 1st-class modules	–	–	–	+
impure abstraction	–	–[1]	+	+
safe module equivalence	–[2]	–	–	–
no loss of type equivalences	–	+	+	+

[1] Generative functors can be turned applicative after the fact

[2] Though you have to try harder

Example

```
module Set = fun Elem : ORD ⇒ {
  type elem = Elem.t
  type set = list elem
  val empty = []
  val add x s = case s of
    | [] ⇒ [x]
    | y :: s' ⇒ if not (Elem.leq x y) then y :: add x s'
                 else if Elem.leq y x then s
                 else x :: s
  val mem x s = case s of
    | [] ⇒ false
    | y :: s' ⇒ Elem.leq y x and (Elem.leq x y or mem x s')
} :> SET where type elem = Elem.t
```