Why Applicative Functors Matter

Derek Dreyer
Toyota Technological Institute at Chicago

WG2.8 Meeting, Iceland
July 16-20, 2007
Matthias Felleisen Gets In a Good Jab

• Matthias Felleisen, POPL PC Chair report, Jan. 2007
  – In a harangue about the navel-gazing nature of POPL
  – Lists a variety of issues that people write POPL papers about, which are obscure to the outside world
  – The list includes “applicative vs. generative functors”

• I feel a twinge in my heart, for:
  – This is what most of my papers are (at least tangentially) about
  – But Matthias has a point
Applicative vs. Generative Functors

• Despite many papers on the ML module system,
  – Few people (even in the POPL/FP community) understand or care what the distinction is all about.

• Reasons for this:
  – It’s not clear what the distinction is all about, from a practical programming standpoint.
  – Yes, applicative functors allow more programs to typecheck, but their motivations are somewhat weak.
The Point of This Talk

- Applicative functors DO matter!
  - But not for the reasons usually given (Leroy, POPL’95)

- In this talk:
  - An overview of traditional semantics and motivations for applicative functors, and why I don’t buy them
  - A new “killer app” for applicative functors
Why Generative Functors Matter

• Data encapsulation

• Need to tie abstract types to piece of \textit{mutable state} that is only created dynamically
  – Canonical example: The SymbolTable functor

• Very similar motivation to ownership types in the OO world

• I won’t talk about them anymore today (until the very end)
Fully Transparent
Higher-Order Functors

• Canonical example: The Apply functor
  – signature SIG = sig type t … end
  – functor Apply (F : SIG → SIG) (X : SIG) = F(X)
  – Apply : (SIG → SIG) → (SIG → SIG)

• Problem:
  – Apply(F) does not have the same signature as F.
  – E.g. functor Id (X : SIG) = X
  – Id : (X : SIG) → SIG where type t = X.t
  – Apply(Id) : SIG → SIG
Applicative Functors to the Rescue

- \( \text{Apply} : (F : \text{SIG} \rightarrow \text{SIG}) \rightarrow (X : \text{SIG}) \rightarrow \text{SIG} \text{ where type } t = F(X).t \)
- \( \text{Apply (Id)} : (X : \text{SIG}) \rightarrow \text{SIG} \text{ where type } t = \text{Id}(X).t \)
i.e. \( (X : \text{SIG}) \rightarrow \text{SIG} \text{ where type } t = X.t \)
Weak Motivation

- Apply : (F : SIG → SIG) →
  (X : SIG) → SIG where type t = \( F(X).t \)

- Apply (Id) : (X : SIG) → SIG where type t = Id(X).t
  i.e. (X : SIG) → SIG where type t = X.t

- Great, but who cares about the Apply functor?
  - It’s a very lame functor.
  - Other, more exciting, examples have not been forthcoming.
  - At least that’s my impression, but maybe Xavier or Norman would beg to differ?
Identifying Equivalent Functor Instantiations

- Canonical example: The MkSet functor
  - signature ORD = sig type t; val cmp : t → t → bool end
  - signature SET = sig type t; type elem;
    val empty : t;
    val insert : elem → t → t ...
  end
  - functor MkSet (X : ORD)
    :> SET where type elem = X.t
    = struct ... end
Identifying Equivalent Functor Instantiations

• Canonical example: The MkSet functor
  – structure OrdInt = struct type t = int; val cmp = …end
  – structure IntSet1 = MkSet(OrdInt)
  – structure IntSet2 = MkSet(OrdInt)
  – In SML, IntSet1.t ≠ IntSet2.t, but they are in fact compatible types.
  – In OCaml, IntSet1.t = MkSet(OrdInt).t = IntSet2.t.
Identifying Equivalent Functor Instantiations

• My assessment:
  – OCaml’s behavior (on this example) is appealing.
  – But doesn’t seem critically important.

• Moreover, we run into the *module equivalence* problem:
  – What is the right way to compare M and N when checking whether F(M).t = F(N).t?
The Module Equivalence Problem

- In OCaml, equivalence of types of the form $F(X).t$ is purely *syntactic*. Example:
  - structure IntSet = MkSet(OrdInt)
  - structure MyOrdInt = OrdInt
  - structure MyIntSet = MkSet(MyOrdInt)
  - $\text{MyIntSet}.t \neq \text{IntSet}.t$ because
    $\text{MkSet(MyOrdInt)}.t \neq \text{MkSet(OrdInt)}.t$ syntactically.

- This makes applicative functor semantics very brittle.
The Module Equivalence Problem

• Subsequent papers on the ML module system attempted to address this by instead comparing functor arguments via “static equivalence” (Shao 99, Russo 00, Dreyer et al. 03)
  – Two modules are *statically equivalent* if their type components are equal.
  – This equates MyOrdInt and OrdInt, and thus MyIntSet.t and IntSet.t, as we desired.
  – But it also equates too many other things:
    • OrdIntLt = OrdIntGt, statically
    • MkSet(OrdIntLt).t ≠ MkSet(OrdIntGt).t, in principle
The Module Equivalence Problem

• Really what we want is *contextual equivalence*.
  – MkSet(OrdInt).t = MkSet(OrdInt).t, obviously
  – MkSet(OrdInt).t = MkSet(MyOrdInt).t, since MyOrdInt is just a copy of OrdInt
  – MkSet(OrdIntLt).t ≠ MkSet(OrdIntGt).t, since OrdIntLt ≠ OrdIntGt (contextually)

• However, contextual equivalence is undecidable.
  – I’ll return to this issue toward the end of the talk.
Summary

• Given just Xavier’s original motivations, I don’t think applicative functors are worth it.

• One motivation Xavier did not present, but that I used to find very compelling, is *recursive modules*.
  – Encoding data structures such as “bootstrapped heaps” seems to require applicative functors.
  – But my work on “recursive type generativity” (ICFP ’05, ’07) shows how to support such data structures just fine with generative functors.
Modular Type Classes

- POPL ’07: Joint work with Harper and Chakravarty

- Basic idea: Model Haskell type classes using modules
  - Classes are signatures
  - Instances are structures
  - Generic instances are functors
  - Instances do not have global scope, they may be adopted as “canonical” within a local scope
Classes and Instances in ML

signature EQ = sig
    type t
    val eq : t -> t -> bool
end

structure EqInt : EQ = struct
    type t = int
    val eq = Int.eq
end

functor EqProd (X : EQ) (Y : EQ) : EQ = struct
    type t = X.t * Y.t
    fun eq (x1,y1) (x2,y2) =
        (X.eq x1 x2) andalso (Y.eq y1 y2)
end

• Great, but now how do we create the \texttt{eq} function?
Creating an Overloaded Function

• We employ an overload mechanism:

\[
\text{val eq = overload eq from EQ}
\]

• This creates a “polymorphic value” `eq`, represented internally (in the semantics) as an implicit functor:

\[
eq : (X : \text{EQ}) \Rightarrow X.t \rightarrow X.t \rightarrow \text{bool}
\]

• Analogous to Haskell’s qualified types:

\[
eq :: (\text{Eq} \ a) \Rightarrow a \rightarrow a \rightarrow \text{bool}
\]
Making an Instance Canonical

- Designate \texttt{EqInt} and \texttt{EqProd} as canonical in a certain scope:
  ```
  using EqInt, EqProd in
  ```
  ```
  ... 
  ```
Making an Instance Canonical

• Now if we apply \texttt{eq} in that scope:

\begin{verbatim}
using EqInt, EqProd in
...eq (2,3) (4,5)...
\end{verbatim}

• Then the above code typechecks and translates internally to:

\begin{verbatim}
...Val(eq(EqProd(EqInt)(EqInt))) (2,3) (4,5)...
\end{verbatim}

• Similar to evidence translation in Haskell:
  – Here we use modules as evidence
Restrictions on Instance Functors

- Instance functor bodies must be pure and terminating.
  - Important to ensure that references to variables (like eq) do not engender arbitrary effects.

- Instance functors must be transparent. Why?
  - For simplicity, we only supported generative functors.
  - So if a generative functor is not transparent, every application has the effect of creating new abstract types.
  - So a whole class of functors, like MkSet, can’t be used as instance functors.
My Claims

• Applicative functors can increase the expressiveness of modular type classes, bringing them closer to Haskell in a clean and elegant way.

• Making the same purity restriction on applicative functors that we make on instance functors will give us “true” applicative functors, which is what we want anyway.
Motivating Example

• Here is a function `singleton`, that takes an argument `x` and returns the singleton set `{x}`.

```plaintext
val empty = overload empty from SET
val insert = overload insert from SET

fun singleton x = insert x empty

val singleton : (X : SET) => X.elem -> X.t
```
Motivating Example

- But applying this singleton function

\begin{verbatim}
val S = singleton 3
\end{verbatim}

does not actually \textit{per se} create a set. It just elaborates to

\begin{verbatim}
val S : X.t = X.insert 3 X.empty
\end{verbatim}

where \(X\) is bound by a residual constraint

\begin{verbatim}
X : SET where type elem = int
\end{verbatim}
Motivating Example

• If all we have are generative functors, then we have to define the Set module for each type individually.

```plaintext
structure IntSet = MkSet(OrdInt)
using IntSet in
val IS : IntSet.t = singleton 3

structure StrSet = MkSet(OrdString)
using StrSet in
val SS : StrSet.t = singleton "hi"
```

• This is quite cumbersome. Aren’t modular type classes supposed to apply your functors for you?
Applicative Functors to the Rescue

• If MkSet is applicative, then it can be given a transparent signature:

\[(X : \text{ORD}) \rightarrow \text{SET} \text{ where type elem} = X.t \text{ and type } t = \text{MkSet}(X).t\]

• So we can use it as an instance functor:

```ml
using \text{MkSet},\text{OrdInt},\text{OrdString} \text{ in }

\text{val IS} : \text{MkSet}(\text{OrdInt}).t = \text{singleton } 3 \\
\text{val SS} : \text{MkSet}(\text{OrdString}).t = \text{singleton } "\text{hi}"```

Applicative Functors to the Rescue

• Or, better yet:

```haskell
using MkSet in
  val mysingleton = singleton
  : (X : ORD) => X.t -> MkSet(X).t

using OrdInt, OrdString in
  val IS = mysingleton 3
  val SS = mysingleton "hi"
```

• This is an improvement, but may not be quite what we want.
Module Overloading

- We really would like to project directly from MkSet itself:

```haskell
fun mysingleton x = MkSet.insert x MkSet.empty
(* : (X : ORD) => X.t -> MkSet(X).t *)
```

- The type of `mysingleton` may now be inferred.
- The idea is very natural:
  - Just as `eq` is a functor representing an overloaded term, `MkSet` is a functor representing an overloaded structure.
Semantics of Module Overloading

• Treat projection from a functor as a composition of projection and the functor:

\[ "F.\ell" \overset{\text{def}}{=} (\ell) \circ F \]

• “MkSet.insert” = \( \lambda \ (X : \text{ORD}). \text{MkSet}(X).\text{insert} \)
• “MkSet.t” = \( \lambda \ (X : \text{ORD}). \text{MkSet}(X).t \)
• For MkSet.t, argument cannot be inferred
• Projection and application of a functor commute:

\[ F(M).\ell = F.\ell(M) \]
No Need for overload

• No more need for the overload mechanism
• Overloading is just projection from the identity functor:
  – functor EQ (X : EQ) = X
  – “EQ.eq” is the overloaded “eq” operator (i.e. functor)
  – “open EQ” introduces “eq” into scope directly
Type Operators

• MkSet.t is a functor mapping an ORD module to a type.
• We have \( \text{MkSet.t(OrdIntLt)} \neq \text{MkSet.t(OrdIntGt)} \).
• But say OrdInt has been “used” as the canonical implementation of ORD at int. Then, we’d like to write \( \text{MkSet.t(int)} \) and have that mean \( \text{MkSet.t(OrdInt)} \).
• Solution:
  - \( \text{Set} = \lambda (\alpha). \lambda (X : \text{ORD where type } t = \alpha). \text{MkSet}(X) \)
  - \( \text{Set.t(int)} \) elaborates to \( \text{Set.t(int)(OrdInt)} \) when used in a type expression in the scope of “using OrdInt”.

Another Example

• Assume Set functor has fromList and toList functions,
  val crossList : α list → β list → (α × β) list

• fun crossSet S1 S2 =
  Set.fromList (crossList (Set.toList S1) (Set.toList S2))

• val crossSet :
  (X:ORD, Y: ORD) ⇒
  MkSet(X).t → MkSet(Y).t → MkSet(OrdProd(X)(Y)).t
Another Example

• Assume Set functor has fromList and toList functions,
  val crossList : α list → β list → (α × β) list

• fun crossSet S1 S2 =
  Set.fromList (crossList (Set.toList S1) (Set.toList S2))

• val crossSet :
  (X:ORD, Y: ORD) ⇒
  Set.t (X.t) → Set.t (Y.t) → Set.t (X.t × Y.t)
Conclusion #1

Modular type classes

= 

“Killer app” for applicative functors
Contextual Module Equivalence

- One of the original problems with applicative functors:
  - Want to compare functor args via contextual equivalence

- A conservative form of contextual module equivalence can be implemented via static equivalence:
  - At every value binding, define a hidden ADT “rep”
  - If two values have the same hidden “rep” type, then one must be a copy of the other
  - So static equivalence $\Rightarrow$ contextual equivalence
This trick DOES NOT work if applicative functors can have impure bodies. Example:
- \( F = \lambda () \). struct val x = ref 3 end
- \( A = F(), B = F() \)
- \( A.x.rep = B.x.rep, \) but \( A.x.val \neq B.x.val \)

More generally, this trick only works if:
- \( X = Y \) \( \Rightarrow \) \( F(X) = F(Y) \)
- I.e. \( F \) is a “true” applicative functor
Conclusion #2

“True” applicative functors

The way to go
Questions for the Crowd

• Are there any good uses of impure applicative functors?
  – I think so, but they are not very compelling.

• Are there any good uses of pure generative functors?
  – I don’t think so.

• My current thinking:
  – Applicative/Generative = Pure/Impure, plain and simple.
Thank you!