Lightweight higher-kinded polymorphism
(Extended version)

Jeremy Yallop and Leo White

University of Cambridge

Abstract. Higher-kinded polymorphism —i.e. abstraction over type *constructors*— is an essential component of many functional programming techniques such as monads, folds, and embedded DSLs. ML-family languages typically support a form of abstraction over type constructors using functors, but the separation between the core language and the module language leads to awkwardness as functors proliferate.
We show how to express higher-kinded polymorphism in OCaml without functors, using an abstract type `app` to represent type application, and opaque brands to denote abstractable type constructors. We demonstrate the flexibility of our approach by using it to translate a variety of standard higher-kinded programs into functor-free OCaml code.

1 Introduction

Polymorphism abstracts types, just as functions abstract values. Higher-kinded polymorphism takes things a step further, abstracting both types and type constructors, just as higher-order functions abstract both first-order values and functions.

Here is a function with a higher-kinded type. The function `when` conditionally executes an action:

```
when b m = if b then m else return ()
```

In Haskell, `when` receives the following type:

```
when :: ∀ (m :: ∗ → ∗). Monad m ⇒ Bool → m () → m ()
```

The kind ascription $* \to *$ makes explicit the fact that `m` is a *higher-kinded* type variable: it abstracts type constructors such as `Maybe` and `[]`, which can be applied to types such as `Int` and `()` to build new types. The type of `when` says that its second argument and return value are monadic computations returning `()`, but the monad itself is not fixed: `when` can be used at any type `m ()` where `m` builds a type from a type and is an instance of the `Monad` class.

In contrast, in OCaml, as in other ML-family languages, all type variables have kind $*$. In order to abstract a type constructor one must use a *functor*. Here is an implementation of `when` in OCaml:

```
module When (M : Monad) = struct
  let f b m = if b then m else M.return ()
```

```
end
```

The `When` functor receives the following type:

```
module When (M : Monad) :sig
  val f : bool → unit M.t → unit M.t
end
```

Defining `When` is more work in OCaml than in Haskell. For callers of `When` the difference is even more pronounced. Here is a Haskell definition of `unless` using when:

```
unless b m = when (not b) m
```

Defining `Unless` in OCaml involves binding three modules. First, we define a functor to abstract the monad once more, binding both the functor and its argument. Next, we instantiate the `When` functor with the monad implementation and bind the result. Finally, we can call the function:

```
module Unless(M : Monad) = struct
  module W = When(M)
  let unless b m = W.f (not b) m
end
```

The situation is similar when we come to use our functions at a particular monad. We must first instantiate `When` or `Unless` with a module satisfying the `Monad` interface before we can use it to build computations. The following example instantiates `Unless` with a module implementing the state monad, then uses the result to build a computation that conditionally writes a value:

```
let module U = Unless(StateM) in
  U.unless (v < 0) (StateM.put v)
```

Why does OCaml require us to do so much work to define such simple functions? One issue is the lack of overloading: in order to use functions like `when` with multiple monads we must explicitly pass around dictionaries of functions. However, most of the syntactic heaviness comes from the lack of higher-kinded polymorphism: functors are the only mechanism ML provides for abstracting over type constructors. The purpose of this paper is to address this second issue, bringing higher-kinded polymorphism into the core OCaml language, and making it almost as convenient to define `when` and `unless` in OCaml as in Haskell.

1.1 The alias problem

At this point the reader might wonder why we do not simply adopt the Haskell approach of adding higher-kinded polymorphism directly to the core language. The answer lies in a fundamental difference between type constructors in Haskell and type constructors in OCaml.

In Haskell data and newtype definitions create fresh data types. It is possible to hide the data constructors of such types by leaving them out of the export list of the defining module, but the association between a type name and the data type it denotes cannot be abstracted. It is therefore straightforward for the type checker to determine whether two type names denote the same data type: after expanding synonyms, type names denote the same data types exactly when the names themselves are the same.

OCaml provides more flexible mechanisms for creating abstract types. An entry `type t` in a signature may hide either a fresh data type definition such as `type t = T of int` or as an alias such as `type t = int`. Abstracting types with signatures is sometimes only temporary, since instantiating a functor can replace abstract types in the argument signature with concrete representations. Checking whether two type names denote the same data type is therefore a more subtle matter in OCaml than in Haskell, since abstract types with no visible equalities may later turn out to be equal after all.

Since OCaml cannot distinguish between data types and aliases, it must support instantiating type variables with either. This works well for type variables of base kind, but breaks down with the addition of higher-kinded type variables. To see the difficulty, consider the unification of the following pair of type expressions

```
'a 'f   ~   (int * int) list
```

where `'f` is a higher-kinded type variable. If there are no other definitions in scope then there is an obvious solution, unifying `'a` with `(int * int)` and `'f` with `list`. Now suppose that we also have the following type aliases in scope:

```
type 'a plist = ('a * 'a) list
type 'a iplist = (int * int) list
```

With the addition of `plist` and `iplist` there is no longer a most general unifier. Unifying `'f` with either `plist` or `iplist` gives two new valid solutions, and none of the available solutions is more general than the others.

One possible response to the loss of most general unifiers is to give up on type inference for higher-kinded polymorphism. This is the approach taken by OCaml's functors, which avoid ambiguity by explicitly annotating every instantiation. We will now consider an alternative approach that avoids the need to annotate instantiations, bringing higher-kinded polymorphism directly into the core language.

1.2 Defunctionalization

Since we cannot use higher-kinded type variables to represent OCaml type constructors, we are faced with the problem of abstracting over type expressions of higher kind in a language where all type variables have base kind. At first sight the problem might appear intractable: how can we embed an expressive object language in a less expressive host language?

Happily, there is a well-understood variant of this problem from which we can draw inspiration. Four decades ago John Reynolds introduced *defunctionalization*, a technique for translating higher-order programs into a first-order language [Reynolds, 1972].

The following example illustrates the defunctionalization transform. Here is a higher-order ML program which computes a sum and increments a list of numbers:

```
let rec fold : type a b. (a * b → b) * b * a list → b =
 fun (f, u, l) → match l with
  | [] → u
  | x :: xs → f (x, fold (f, u, xs))

let sum l = fold ((fun (x, y) → x + y), 0, l)
let add (n, l) = fold ((fun (x, l') → x + n :: l'), [], l)
```

Defunctionalizing this program involves introducing a datatype `arrow` with two constructors, one for each of the two function terms; the arguments to each constructor represent the free variables of the corresponding function term, and the type parameters to `arrow` represent the argument and return types of the function. We follow Pottier and Gauthier [2004] in defining `arrow` as a generalised algebraic data type (GADT), which allows the instantiation of the type parameters to vary with each constructor, and so makes it possible to preserve the well-typedness of the source program.

```
type (_, _) arrow =
  Fn_plus : ((int * int), int) arrow
| Fn_plus_cons : int → ((int * int list), int list) arrow
```

The second step introduces a function, `apply`, that relates each constructor of `arrow` to the function body.

```
let apply : type a b. (a, b) arrow * a → b =
  fun (appl, v) → match appl with
   | Fn_plus → let (x, y) = v in x + y
   | Fn_plus_cons n → let (x, l') = v in x + n :: l'
```

We can now replace function terms with constructors of `arrow` and indirect calls with applications of `apply` to turn the higher-order example into a first order program:

```
let rec fold : type a b. (a * b, b) arrow * b * a list → b =
 fun (f, u, l) → match l with
  | [] → u
  | x :: xs → apply (f, (x, fold (f, u, xs)))

let sum l = fold (Fn_plus, 0, l)
let add (n, l) = fold (Fn_plus_cons n, [], l)
```

1.3 Type defunctionalization

Defunctionalization transforms a program with higher-order values into a program where all values are first-order. Similarly, we can change a program with higher-kinded type expressions into a program where all type expressions are of kind $*$, the kind of types.

The first step is to introduce an abstract type constructor, analogous to `apply`, for representing type-level application:

```
type ('a, 'f) app
```

OCaml excludes higher-kinded type expressions syntactically by requiring that the type operator be a concrete name: `'a list` is a valid type expression, but `'a 'f` is not. The `app` type sidesteps the restriction, much as the `apply` function makes it possible to embed the application of a higher-order function in a first-order defunctionalized program. The type expression `(s, t) app` represents the application of the type expression `t` to the type expression `s`. We can now abstract over type constructors by using a type variable for the operator term `t`.

Eliminating higher-order functions associates a constructor of `arrow` with each function expression from the original program. In order to eliminate higher-kinded type expressions we associate each type expression with a distinct instantiation of `app`. More precisely, for each type constructor `t` which we wish to use in a polymorphic context we introduce an uninhabited opaque type `T.t`, called the *brand*. Brands appear as the operator argument to `app`; for example, we can represent the type expression `'a list` as `('a, List.t) app`, where `List.t` is the brand for `list`. With each brand we associate injection and projection functions for moving between the concrete type and the corresponding instantiation of `app`:

```
module List : sig
  type t
  val inj : 'a list → ('a, t) app
  val prj : ('a, t) app → 'a list
end
```

We now have the operations we need to build and call functions that abstract over type constructors. Here is a second OCaml implementation of the `when` function from the beginning of this paper:[1]

```
let when_ (d : _ #monad) b m = if b then m else d#return ()
```

The first parameter `d` is a dictionary of monad operations analogous to the type class dictionary passed to `when` in a typical implementation of Haskell [Wadler and Blott, 1989]. (We defer further discussion of dictionary representation to Section 2.3.) Our earlier implementation received the dictionary as a functor argument in order to accommodate abstraction over the type constructor, but the introduction of `app` makes it possible to write `when` entirely within the core language. This second implementation of `when` receives the following type:

```
val when_ : 'm #monad → bool → (unit, 'm) app → (unit, 'm) app
```

[1] We append an underscore to variable names where they clash with OCaml keywords.

```
type ('a, 'f) app

module type Newtype1 = sig
  type 'a s
  type t
  val inj : 'a s → ('a,t) app
  val prj : ('a,t) app → 'a s
end

module Newtype1(T : sig type 'a t end):
  Newtype1 with type 'a s = 'a T.t
```

Fig. 1. The *higher* interface

```
module type Newtype2 = sig
  type ('a, 'b) s
  type t
  val inj : ('a,'b) s → ('b,('a,t) app) app
  val prj : ('b,('a,t) app) app → ('a,'b) s
end

module Newtype2(T : sig type ('a,'b) t end):
  Newtype2 with type ('a,'b) s = ('a,'b) T.t
```

Fig. 2. The Newtype2 functor

The improvement becomes even clearer when we implement `unless` without a functor in sight:

```
let unless d b m = when_ d (not b) m
```

There is a similar improvement when using `when` and `unless` at particular monads. Once again we find that we no longer need to instantiate a functor, since the dictionary parameter is passed as a regular function argument. Here is our earlier example that conditionally writes a value in the state monad, adapted to our new setting:

```
unless state (v < 0) (state#put v)
```

2 The interface

We have written a tiny library called *higher* to support programming with `app`. Figure 1 shows the interface of the *higher* library.[2] The `Newtype1` functor generates brands together with their associated injection and projection functions, preserving the underlying concrete type under the name `s` for convenience. For example, applying `Newtype1` to a structure containing the concrete `list` type gives the `List.t` brand from Section 1.3.

```
module List = Newtype1(struct type 'a t = 'a list end)
```

In fact, as the numeric suffix in the `Newtype1` name suggests, *higher* exports a family of functors for building brands. Figure 2 gives another instance, for concrete types with two parameters. However, rather than introducing a second version of `app` to accompany `Newtype2`, we use `app` in a curried style. One of the benefits of higher kinded polymorphism is the ability to partially apply multi-parameter type constructors, and the currying in `Newtype2` makes this possible in our setting.

[2] The *higher* library is available on opam: `opam install higher`

The remainder of this section shows how various examples from the literature can be implemented using *higher*.

2.1 Example: higher-kinded folds

Higher-kinded polymorphism was introduced to Haskell to support constructor classes such as `Monad` [Jones, 1995, Hudak et al., 2007]. However, not all uses of higher kinds involve constructor classes. Traversals of non-regular datatypes (whose definitions contain non-trivial instantiations of the definiendum) typically involve higher-kinded polymorphism. Here is an example: the type `perfect` describes perfectly balanced trees, with 2^n elements:

```
type 'a perfect = Zero of 'a | Succ of ('a * 'a) perfect
```

A fold over a `perfect` value is parameterised by two functions, `zero`, applied at each occurrence of `Zero`, and `succ`, applied at each occurrence of `Succ`. In diagram form the fold has the following simple shape:

$$\begin{array}{cccccccc}
\texttt{Succ} & (\texttt{Succ} & \ldots & (\texttt{Succ} & (\texttt{Zero} & \texttt{v})) & \ldots)\\
\downarrow & \downarrow & & \downarrow & \downarrow & & \\
\texttt{succ} & (\texttt{succ} & \ldots & (\texttt{succ} & (\texttt{zero} & \texttt{v})) & \ldots)
\end{array}$$

What distinguishes this fold from a similar function defined on a regular datatype is that each occurrence of `Succ` is used at a different type. If the outermost constructor builds an `int perfect` value then the next constructor builds an `(int * int) perfect`, the next an `((int * int) * (int * int)) perfect`, and so on. For maximum generality, therefore, we must allow the types of `zero` and `succ` to vary in the same way.[3] In Haskell we might define `foldp` as follows:

```
foldp :: (∀a. a → f a) → (∀a. f (a, a) → f a) → Perfect a → f a
foldp zero succ (Zero l) = zero l
foldp zero succ (Succ p) = succ (foldp zero succ p)
```

Here is a corresponding definition in OCaml, using a record type with polymorphic fields for the higher-rank types (nested quantification) and using `app` to introduce higher-kinded polymorphism:

```
type 'f perfect_folder = {
  zero: 'a. 'a → ('a, 'f) app;
  succ: 'a. ('a * 'a, 'f) app → ('a, 'f) app;
}

let rec foldp : 'f 'a. 'f perfect_folder → 'a perfect → ('a, 'f) app =
   fun { zero; succ } → function
   | Zero l → zero l
   | Succ p → succ (foldp { zero; succ } p)
```

[3] Hinze [2000] shows how to take generalization of folds over nested types significantly further than the implementation we present here.

```
type ('a, 'b) eq

val refl : unit → ('a, 'a) eq

module Subst (F : sig type 'a f end):
sig
 val subst : ('a, 'b) eq →
  'a F.f → 'b F.f
end
```

Fig. 3. Leibniz equality without *higher*

```
module Eq : Newtype2
type ('a, 'b) eq = ('b, ('a, Eq.t) app) app

val refl : unit → ('a, 'a) eq



val subst : ('a, 'b) eq →
  ('a, 'f) app → ('b, 'f) app
```

Fig. 4. Leibniz equality with *higher*

The `foldp` function has a number of useful properties. A simple one, immediately apparent from the diagram, is that `foldp Zero Succ` is the identity. In order to instantiate the result type we need a suitable instance of `app`, which we can obtain using `Newtype1`.

```
module Perfect = Newtype1(struct type 'a t = 'a perfect end)
```

Passing `Zero` and `Succ` requires a little massaging with `inj` and `prj`.

```
let idp p = Perfect.(prj (foldp { zero = (fun l → inj (Zero l));
                                  succ = (fun b → inj (Succ (prj b)))} p))
```

It is easy to verify that `idp` implements the identity function.

2.2 Example: Leibniz equality

Our second example involves higher-kinded polymorphism in the definition of a datatype. As part of a library for dynamic typing, Baars and Swierstra [2002] introduce the following definition of type equality:

```
newtype Equal a b = Equal (∀ (f :: * → *). f a → f b)
```

The variable `f` abstracts over one-hole type contexts — type expressions which build a type from a type. The types encode Leibniz's law that `a` and `b` can be considered equal if they are interchangeable in any context `f`. A value of type `Equal a b` serves both as proof that `a` and `b` are equal and as a coercion between contexts instantiated with `a` and `b`. Ignoring $\perp$ values, there is a single inhabitant of `Equal`, the value `Equal id` of type `Equal a a`, which serves as a proof of equality between any type `a` and itself.

Yallop and Kiselyov [2010] show how first-class modules make it possible to define an OCaml type `eq` equivalent to `Equal`. A minimised version of `eq` and its core operations is given in Figure 3. There are two operations: `refl` introduces the sole inhabitant, a proof of reflexive equality, and `subst` turns an equality proof into a coercion within any context `f`.

```
class virtual ['m] monad : object
  method virtual return : 'a. 'a → ('a, 'm) app
  method virtual bind : 'a 'b. ('a, 'm) app → ('a → ('b, 'm) app) → ('b, 'm) app
end
```

Fig. 5. The monad interface in OCaml

```
type ('a, 'f) free = Return of 'a | Wrap of (('a, 'f) free, 'f) app
module Free = Newtype2(struct type ('a, 'f) t = ('a, 'f) free end)
```

Fig. 6. The free monad data type in OCaml

Figure 4 gives a second definition of `eq` and its operations using *higher*. As with `unless`, using the functor version of Figure 3 is significantly heavier than the *higher* version of Figure 4. Here is a definition of the transitive property of equality using the implementation of Figure 3:

```
let trans : type a b c. (a, b) eq → (b, c) eq → (a, c) eq =
  fun ab bc →
    let module S = Subst(struct type 'a f = (a, 'a) eq end) in
    S.subst bc ab
```

And here is a definition using *higher*:

```
let trans ab bc = subst bc ab
```

Both implementations receive the same type:

```
val trans: ('a, 'b) eq → ('b, 'c) eq → ('a, 'c) eq
```

The contrast between the implementations of `refl` and `subst` is similarly striking. The interested reader can find the full implementations in Appendix A.

2.3 Example: the codensity transform

Much of the appeal of higher-kinded polymorphism arises from the ability to define overloaded functions involving higher-kinded types. Constructor classes [Jones, 1995] turn monads (and other approaches to describing computation such as arrows [Hughes, 2000] and applicative functors [Mcbride and Paterson, 2008]) from design patterns into named program entities. The `Monad` interface requires abstraction over type constructors, and hence higher kinds, but defining it brings a slew of benefits: it becomes possible to build polymorphic functions and notation which work for any monad, and to construct a hierarchy of related interfaces such as `Functor` and `MonadPlus`.

OCaml does not currently support overloading, making many programs which find convenient expression in Haskell cumbersome to write. However, the loss of elegance does not arise from a loss of expressive power: although type classes are unavailable we can achieve similar results by programming directly in the target

```
let monad_free (functor_free : 'f #functor_) = object
  inherit [('f, Free.t) app] monad
  method return v = Free.inj (Return v)
  method bind =
    let rec bind m k = match m with
      | Return a → k a
      | Wrap t → Wrap (functor_free#fmap (fun m → bind m k) t) in
    fun m k → Free.inj (bind (Free.prj m) (fun a → Free.prj (k a)))
end
```

Fig. 7. The free monad instance in OCaml

language of the translation which eliminates type classes in favour of dictionary passing [Wadler and Blott, 1989]. We might reasonably view these explicit dictionaries as temporary scaffolding that will vanish once the plans to introduce overloading to OCaml come to fruition [Chambart and Henry, 2012].

We now turn to an example of a Haskell program that makes heavy use of higher-kinded overloading. The *codensity transform* [Voigtländer, 2008] takes advantage of higher-kinded polymorphism to systematically substitute more efficient implementations of computations involving free monads, leading to asymptotic performance improvements. We will focus here on the constructs necessary to support the codensity transform rather than on the computational content of the transform itself, which is described in Voigtländer's paper. The code in this section is not complete (the definitions of `abs`, `C`, and `functor_` are missing), but we give a complete translation of the code from Voigtländer [2008, Sections 3 and 4] in Appendix B.

Figure 5 shows the `monad` interface in OCaml. We represent a type class by an OCaml virtual class —i.e., a class with methods left unimplemented. The type class variable `m` of type $* \to *$ becomes a type parameter, which is used in the definition of `monad` as an argument to our type application operator `app`.

Figure 6 defines the free monad type [Voigtländer, 2008, Section 3]. The use of `app` in the definition of `free` reflects the fact that the type parameter `'f` has higher kind; without *higher* we would have to define the `free` within a functor.

Figure 7 gives the free monad instance over a functor using the `free` type. We represent type class instances in OCaml as values of object type. Instantiating and inheriting the `monad` class provides type checking for `return` and `bind`. Constraints in the instance definition in the Haskell code become arguments to the function; our definition says that `('f, Free.t) app` is an instance of `monad` if `'f` is an instance of `functor`.

Figure 8 defines the `freelike` interface. In Voigtländer's presentation `FreeLike` is a multi-parameter type class with two superclasses. In our setting the parameters become type parameters of the virtual class and the superclasses become class arguments which must be supplied at instantiation time. We bind the class arguments to methods so that we can easily retrieve them later.

```
class virtual ['f, 'm] freelike (pf : 'f functor_) (mm : 'm monad) = object
  method pf : 'f functor_ = pf      method mm : 'm monad = mm
  method virtual wrap : 'a. (('a, 'm) app, 'f) app → ('a, 'm) app
end
```

Fig. 8. The freelike interface in OCaml

```
type ('a, 'f) freelike_poly = {
  fl: 'm 'd. (('f, 'm) #freelike as 'd) → ('a, 'm) app
}

let improve (d : _ #functor_) { fl } =
   Free.prj (abs (monad_free d) (C.prj (fl (freelike_c d (freelike_free d)))))
```

Fig. 9. The improve function in OCaml

```
improve :: Functor f ⇒ (∀m. FreeLike f m → m a) → Free f a
improve m = abs m
```

Fig. 10. The improve function in Haskell

Figure 9 shows the `improve` function, the entry point to the codensity transform. In Haskell `improve` has a concise definition (Figure 10) due to the amount of work done by the type class machinery; in OCaml we must perform the work of building and passing dictionaries ourselves. As in a previous example (Section 2.1) we use a record with a polymorphic field to introduce the necessary higher-rank polymorphism.

Appendix C gives a complete implementation of the codensity transform, and a translation of Voigtländer's example which applies it to an *echo* computation.

2.4 Example: kind polymorphism

Standard Haskell's kind system is "simply typed": the two kind formers are the base kind $*$ and the kind arrow $\rightarrow$, and unknown types are defaulted to $*$. Recent work adds kind polymorphism, increasing the number of programs that can be expressed [Yorgey et al., 2012]. In contrast *higher* lacks a kind system altogether: the brands that represent type constructors are simply uninhabited members of the base kind $*$.

The obvious disadvantage to the lack of a kind system is that the type checker is no help in preventing the formation of ill-kinded expressions, such as `(List.t, List.t) app`. However, this drawback is not so serious as might first appear, since it does not introduce any means of forming ill-typed values, and so cannot lead to runtime errors. In fact, the absence of well-kindedness checks can be used to advantage: it allows us to write programs which require the kind polymorphism extension in Haskell.

```
class virtual ['f] category = object
  method virtual ident : 'a. ('a, ('a, 'f) app) app
  method virtual compose : 'a 'b 'c.
    ('b, ('a, 'f) app) app → ('c, ('b, 'f) app) app → ('c, ('a, 'f) app) app
end
```

Fig. 11. The category interface.

```
module Fun = Newtype2(struct type ('a, 'b) t = 'b → 'a end)
let category_fun = object
  inherit [Fun.t] category
  method ident = Fun.inj id
  method compose f g = Fun.inj (fun x → Fun.prj g (Fun.prj f x))
end
```

Fig. 12. A category instance for →.

```
type ('n, 'm) ip = { ip: 'a. ('a, 'm) app → ('a, 'n) app }
module Ip = Newtype2(struct type ('n, 'm) t = ('n, 'm) ip end)
let category_ip = object
  inherit [Ip.t] category
  method ident = Ip.inj { ip = id }
  method compose f g = Ip.inj {ip = fun x → (Ip.prj g).ip ((Ip.prj f).ip x) }
end
```

Fig. 13. A category instance for index-preserving functions.

Figure 11 defines a class `category` parameterised by a variable `'f`. In the analogous type class definition standard Haskell would give the variable corresponding to `'f` the kind $* \to * \to *$; the polymorphic kinds extension gives it $\forall\kappa.\ \kappa \to \kappa \to *$, allowing the arguments to be type expressions of any kind. Since there is no kind checking in *higher*, we can also instantiate the arguments of `'f` with expressions of any kind. Figure 12 gives an instance definition for $\to$, whose arguments have kind $*$; Figure 13 adds a second instance for the category of index-preserving functions, leaving the kinds of the indexes unspecified.

The extended version of this paper continues the example, showing how *higher* supports higher-kinded non-regularity.

3 Implementations of *higher*

Up to this point we have remained entirely within the OCaml language. Both the interfaces and the examples are written using the current release of OCaml (4.01). However, running the code requires an implementation of the *higher* interface, which requires a small extension to pure OCaml. We now consider two implementations of *higher*, the first based on an unsafe cast and the second based on an extension to the OCaml language.

Let us return to the analogy of Section 1.3. The central point in an implementation of *higher* is a means of translating between values of the `app` family

```
type family Apply f p :: *
newtype App f b = Inj { prj :: Apply f b }

data List
type instance Apply List a = [a]
```

Fig. 14. Implementing *higher* with type families

```
type ('p, 'f) app

module Newtype1 (T : sig type 'a t end) = struct
  type 'a s = 'a T.t
  type t
  let inj : 'a s → ('a, t) app = Obj.magic
  let prj : ('a, t) app → 'a s = Obj.magic
end
```

Fig. 15. Implementing *higher* with an unchecked cast

of types and values of the corresponding concrete types, much as defunctionalization involves translating between higher-order function applications and uses of the `apply` function. However, defunctionalization is a whole program transformation: a single `apply` function handles every translated higher-order call. Since we do not wish to require that every type used with *higher* is known in advance, we need an implementation that makes it possible to extend `app` with new inhabitants as needed.

We note in passing that Haskell's type families [Schrijvers et al., 2008], which define extensible type-level functions, provide exactly the functionality we need. Figure 14 gives an implementation, with a type family `Apply` parameterised by a brand and a type and a type definition `App` with injection and projection functions `Inj` and `Prj`. The `type instance` declaration adds a case to `Apply` that matches the abstract type `List` and produces the representation type [a].

3.1 First implementation: unchecked cast

The first implementation is shown in Figure 15. Each instantiation of the `Newtype1` constructor generates a fresh type `t` to use as the brand. The `inj` and `prj` functions which coerce between the concrete type `'a s` and the corresponding defunctionalized type `('a, t) app` are implemented using the unchecked coercion function `Obj.magic`.

Although we are using an unchecked coercion within the implementation of `Newtype1` the module system ensures that type safety is preserved. Each module to which `Newtype1` is applied generates a fresh brand `t`. Since the only way to create a value of type `('a, t) app` is to apply `inj` to a value of the corresponding type `'a s`, it is always safe to apply `prj` to convert the value back to type `'a s`.

```
type ('p, 'f) app = ..

module Newtype1 (T : sig type 'a t end) () = struct
  type 'a s = 'a T.t
  type t
  type (_, _) app += App : 'a s → ('a, t) app
  let inj v = App v
  let prj (App v) = v
end
```

Fig. 16. Implementing *higher* using open types

3.2 Second implementation: open types

We can avoid the use of an unchecked cast altogether with a small extension to the OCaml language. Löh and Hinze [2006] propose extending Haskell with *open data types*, which lift the restriction that all the constructors of a data type must be given in a single declaration. The proposal is a good fit for OCaml, which already supports a single extensible type for exceptions, and there is an implementation available.[4].

Figure 16 shows an implementation of *higher* using open data types. The ellipsis in the first line declares that `app` is an open data type; each instantiation of the `Newtype1` functor extends `app` with a fresh GADT constructor, `App` which instantiates `app` with the brand `t` and which carries a single value of the representation type `'a s`. The `inj` and `prj` functions inject and project using `App`; although the pattern in `prj` is technically inexhaustive, the fact that the functor generates a fresh `t` for each application guarantees a match in practice.

The empty parentheses in the functor definition force the functor to be generative rather than applicative[5] [Leroy, 1995], so that each application of `Newtype1` generates a fresh type `t`, even if `Newtype1` is being applied to the same argument.

This generative marker is a small deviation from the interface of Figure 1, but essential to ensure that only a single data constructor `App` is generated for each brand `t`. Without the generative marker, multiple applications of `Newtype1` to the same argument would generate modules with compatible brands but incompatible data constructors, leading to runtime pattern-matching failures in `prj`.

4 Related work

We have shown how type defunctionalization can be used to write programs that abstract over OCaml type constructors without leaving the core language. In a

[4] Opam users can install the extended OCaml compiler with the command `opam switch 4.01.0+open-types`.

[5] Explicitly generative functors are a new feature of OCaml, scheduled for the next release: `http://caml.inria.fr/mantis/view.php?id=5905`.

language with features that support case analysis on types, type defunctionalization becomes a yet more powerful tool. Kiselyov et al. [2004] use type defunctionalization together with functional dependencies to support fold operations on heterogeneous lists. Similarly, Jeltsch [2010] implements type defunctionalization using type synonym families to support folds over extensible records.

Kiselyov and Shan [2007] introduce *lightweight static capabilities*, applying phantom types and generativity to mark values as safe for use with an efficient trusted kernel, much as we use generativity in Section 3.1 to ensure the safety of an unchecked cast. Kiselyov and Shan's work is significantly more ambitious than ours; whereas we are interested in expressing programs with higher-kinded polymorphism in ML, they show how to statically ensure properties such as array lengths that were previously thought to require a dependently-typed language. The "brand" terminology is borrowed from Kiselyov and Shan, but their brands are structured type expressions, and significantly more elaborate than the simple atomic names which we use to denote type constructors.

Jones [1995] shows that standard first-order unification suffices for inferring types involving higher-kinded variables so long as the language of constructor expressions has no non-trivial equalities. This insight underlies our use of brands to embed type constructor polymorphism in OCaml.

Swamy et al. [2011] share our aim of reducing the overhead of monadic programming in ML, but take a different approach based on an elaboration of implicitly-monadic ML programs into a language with explicit monad operations. Whereas the present work aims to embed higher-kinded programs into OCaml without changing the language, their proposal calls for significant new support at the language level.

5 Limitations and future work

*The Newtype*N *family* The interface presented in Section 2 consists of a type constructor `app` and a family of functors `Newtype1`, `Newtype2`, ... for extending `app` with new inhabitants. We would ideally like to replace the `Newtype` family with arity-generic operations, but it is unclear whether it is possible to do so in OCaml. For the moment the family of functors seems adequate in practice.

Variance and subtyping Our focus so far has been on expressing higher-kinded programs from Haskell. However, we also plan to explore the interaction of higher-kinded polymorphism with features specific to OCaml. For example, we can obtain a representation of proofs of subtyping by changing the definition of Leibniz equality (Section 2.2) to quantify over positive contexts: a type `a` is a subtype of `b` if it can be coerced to `b` in a positive context (or if `b` can be coerced to `a` in a negative context.) We look forward to exploring the implications of having first-class witnesses of the subtyping relation.

6 Acknowledgements

We are grateful to the anonymous reviewers for their many helpful suggestions.

Bibliography

A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *ICFP '02*, pages 157–166, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

P. Chambart and G. Henry. Experiments in generic programming: runtime type representation and implicit values. OCaml Users and Developers workshop, 2012.

R. Hinze. Efficient generalized folds. In J. Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 1–16, July 2000.

P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(13):67 – 111, 2000. ISSN 0167-6423.

W. Jeltsch. Generic record combinators with static type checking. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 143–154. ACM, 2010.

M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5:1–35, 1 1995. ISSN 1469-7653.

O. Kiselyov and C.-c. Shan. Lightweight static capabilities. *Electron. Notes Theor. Comput. Sci.*, 174(7):79–104, June 2007. ISSN 1571-0661.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 142–153, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199476. URL `http://doi.acm.org/10.1145/199448.199476`.

A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM. ISBN 1-59593-388-3.

C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 89–98, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 15–27, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6.

J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08, pages 388–403, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70593-2.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2.

J. Yallop and O. Kiselyov. First-class modules: hidden power and tantalizing promises. ACM SIGPLAN Workshop on ML, September 2010. Baltimore, Maryland, United States.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5.

A Leibniz equality

This section gives a complete implementation of the definition of Leibniz equality whose core was presented in Section 2.2. The reader is invited to compare the implementation given here with the implementation based on first class modules presented by Yallop and Kiselyov [2010].

A.1 Leibniz equality: interface

```
module Eq : Newtype2
type ('a, 'b) eq = ('b, ('a, Eq.t) app) app
val refl : unit → ('a, 'a) eq
val subst : ('a, 'b) eq → ('a, 'f) app → ('b, 'f) app
val trans : ('a, 'b) eq → ('b, 'c) eq → ('a, 'c) eq
val symm : ('a, 'b) eq → ('b, 'a) eq
val cast : ('a, 'b) eq → 'a → 'b
```

A.2 Leibniz equality: implementation

```
module Id = Newtype1(struct type 'a t = 'a end)

type ('a, 'b) eqaux = { eqaux : 'f. ('a, 'f) app → ('b, 'f) app }
module Eq = Newtype2(struct type ('b, 'a) t = ('a, 'b) eqaux end)
type ('a, 'b) eq = ('b, ('a, Eq.t) app) app

let refl () = Eq.inj { eqaux = fun x → x }
let subst ab ctxt = (Eq.prj ab).eqaux ctxt
let trans ab bc = subst bc ab
let cast eq x = Id.prj (subst eq (Id.inj x))
let symm (type a) eq =
  let module S = Newtype1(struct type 'a t = ('a, a) eq end) in
  S.prj (subst eq (S.inj (refl ())))
```

B The codensity transformation

This section gives a complete implementation of the codensity transform which was partially described in Section 2.3, and a translation of the example illustrating the optimization from Voigtländer's paper. References to the Haskell code corresponding to each definition are given in comments. We refer the reader to Voigtländer [2008] for an exposition of the computational content.

```
(* class Monad *)
class virtual ['m] monad = object
  method virtual return : 'a. 'a → ('a, 'm) app
```

```
  method virtual bind : 'a 'b. ('a, 'm) app → ('a → ('b, 'm) app) → ('b, 'm) app
end

(* class Functor *)
class virtual ['f] functor_ = object
  method virtual fmap : 'a 'b. ('a → 'b) → ('a, 'f) app → ('b, 'f) app
end

(* class (Functor f, Monad m) => Freelike f m *)
class virtual ['f, 'm] freelike (pf : 'f functor_) (mm : 'm monad) = object
  method pf : 'f functor_ = pf
  method mm : 'm monad = mm
  method virtual wrap : 'a. (('a, 'm) app, 'f) app → ('a, 'm) app
end

(* newtype C m a = C { forall b. (a -> m b) -> m b } *)
type ('a, 'm) c = { c : 'b. ('a → ('b, 'm) app) → ('b, 'm) app }
module C = Newtype2(struct type ('a, 'm) t = ('a, 'm) c end)

(* instance Monad (C m) *)
let monad_c () = object
  inherit [('m, C.t) app] monad
  method return a = C.inj {c = fun h → h a }
  method bind =
    let bind = fun { c = p } k → {c = fun h → p (fun a → (k a).c h) } in
      fun m k → (C.inj (bind (C.prj m) (fun a → C.prj (k a))))
end

(* rep :: Monad m => m a -> C m a *)
let rep : 'a 'm. 'm #monad → ('a, 'm) app → ('a, 'm) c =
  fun o m → { c = fun k → o#bind m k }

(* rep :: Monad m => C m a -> m a *)
let abs : 'a 'm. 'm #monad → ('a, 'm) c → ('a, 'm) app =
  fun o c → c.c o#return

(* data Free = Return a | Wrap (f (Free f a)) *)
type ('a, 'f) free = Return of 'a | Wrap of (('a, 'f) free, 'f) app
module Free = Newtype2(struct type ('a, 'f) t = ('a, 'f) free end)

(* instance Functor f => Monad (Free f) *)
let monad_free (functor_free : 'f #functor_) = object
  inherit [('f, Free.t) app] monad
  method return v = Free.inj (Return v)
  method bind =
    let rec bind m k = match m with
      | Return a → k a
      | Wrap t → Wrap (functor_free#fmap (fun m → bind m k) t) in
    fun m k → Free.inj (bind (Free.prj m) (fun a → Free.prj (k a)))
end
```

```
(* instance Functor f => FreeLike (Free f) *)
let freelike_free (ff : 'f #functor_) = object
  inherit ['f,  ('f, Free.t) app] freelike ff (monad_free ff)
  method wrap =
    (* We need the fmap to handle the conversion between ('a, 'f)
       free and the app version in the argument of Wrap *)
    fun x → Free.inj (Wrap (ff#fmap Free.prj x))
end

(* instance FreeLike f m => FreeLike f (C m) *)
let freelike_c (f_functor : 'f #functor_) (freelike : ('f, 'm) #freelike) =
object
  inherit ['f, ('m, C.t) app] freelike f_functor (monad_c ())
  method wrap t =
    C.inj { c = fun h →
      freelike#wrap (f_functor#fmap (fun p → (C.prj p).c h) t)}
end

type ('a, 'f) freelike_poly = {
  fl: 'm 'd. (('f, 'm) #freelike as 'd) → ('a, 'm) app
}

(* improve :: Functor f => (forall m. FreeLike f m => m a) -> Free f a *)
let improve : 'a 'f. 'f #functor_ → ('a, 'f) freelike_poly → ('a, 'f) free =
  fun d { fl } → Free.prj (abs (monad_free d)
                             (C.prj (fl (freelike_c d
                                           (freelike_free d)))))

(* data F_IO b = GetChar (Char -> b) | PutChar Char b *)
type 'b f_io = GetChar of (char → 'b) | PutChar of char * 'b
module F_io = Newtype1(struct type 'b t = 'b f_io end)

(* instance Functor F_IO *)
let functor_f_io = object
  inherit [F_io.t] functor_
  method fmap h l = F_io.inj (match F_io.prj l with
  | GetChar f → GetChar (fun x → h (f x))
  | PutChar (c, x) → PutChar (c, h x))
end

(* getChar :: FreeLike F_IO m => m Char *)
let getChar : 'm. (F_io.t, 'm) #freelike → (char, 'm) app
  = fun f → f#wrap (F_io.inj (GetChar f#mm#return))

(* putChar :: FreeLike F_IO m => Char -> m () *)
let putChar : 'm. (F_io.t, 'm) #freelike → char → (unit, 'm) app
  = fun f c → f#wrap (F_io.inj (PutChar (c, (f#mm#return ()))))
```

```
(* revEcho :: FreeLike F_IO m => m () *)
let rec revEcho : 'm. (F_io.t, 'm) #freelike → (unit, 'm) app
  = fun f →
    let (>>=) c = f#mm#bind c in
    getChar f >>= fun c →
    if (c <> ' ') then
      (revEcho f >>= fun () →
       putChar f c)
    else f#mm#return ()

(* data Output a = Read (Output a) | Print Char (Output a) | Finish a *)
type 'a output = Read of 'a output | Print of char * 'a output | Finish of 'a

(* run :: Free F_IO a -> Stream Char -> Output a *)
let rec run : 'a. ('a, F_io.t) free → char list → 'a output =
  fun f cs → match f with
  | Return a → Finish a
  | Wrap x →
    match F_io.prj x with
    | GetChar f → Read (run (f (List.hd cs)) (List.tl cs))
    | PutChar (c, p) → Print (c, run p cs)

(* run revEcho stream *)
let simulate_original stream =
  run (Free.prj (revEcho (freelike_free functor_f_io)))
    stream

(* run (improve revEcho) stream *)
let simulate_improved stream =
  run (improve functor_f_io { fl = revEcho }) stream
```

C Higher-kinded non-regularity

This section gives an example of higher-kinded non-regularity: a datatype `cat_left` with higher-kinded parameters which are instantiated at a different type in its definition. The `cat_left` type describes computations in the `category` class (Figure 11) represented in left-associative form; the `category` instance for `cat_left` uses the associative and identity laws to rearrange computations in this form.

```
(* The type of category computations in left-associative form. All
   values are of the form

      Compose (Compose(...(Ident, c1), c2), ... cn)
*)
type ('b, 'a, 'f) cat_left =
| Ident : ('a, 'a, 'f) cat_left
| Compose : ('b, 'a, 'f) cat_left * ('c, ('b, 'f) app) app → ('c, 'a, 'f) cat_left
module CL = Newtype3(struct type ('b, 'a, 'f) t = ('b, 'a, 'f) cat_left end)
```

```
(* An instance of category that puts computations into normal form. *)
let category_cat_left (_ : 'f #category) = object (self)
  inherit [('f, CL.t) app] category
  method ident = CL.inj Ident
  method compose : type a b c. (b, (a, ('f, CL.t) app) app) app →
                               (c, (b, ('f, CL.t) app) app) app →
                               (c, (a, ('f, CL.t) app) app) app =
    fun f j → CL.(match prj j with
      Ident → f
    | Compose (g, h) → inj (Compose (prj (self#compose f (inj g)), h)))
end

(* Run a left-associative computation. *)
let rec observe : type f a b. f #category → (b, a, f) cat_left → (b, (a, f) app) app =
  fun cat → function
  | Ident → cat#ident
  | Compose (f, g) → cat#compose (observe cat f) g

(* Lift a value into cat_left. *)
let promote : type f a b. (b, (a, f) app) app → (b, a, f) cat_left =
  fun c → Compose (Ident, c)
```