Arrows as applicatives in a monad

Leo White, Jane Street

Abstract

Arrows are a useful abstraction for representing static computation graphs, but their interface is difficult to use. Arrow notation helps manage this complexity at the cost of requiring users to learn new syntax. We propose a simple interface for arrows built around a type for representing nodes in the computation graph, and a monad for representing programs that build such graphs. The node type is an applicative functor with one adjustment: it's `map` function is an operation of the monad rather than an ordinary function.

1 Arrows are applicatives with sharing

Applicatives[1] and arrows[2] are abstractions that can represent static computation graphs. Figs.1 and 2 show OCaml interfaces for arrows and applicatives[1] We include separate

```
type ('a, 'b) t
val (>>>) :
  ('a, 'b) t -> ('b, 'c) t -> ('a, 'c) t
val arr : ('a -> 'b) -> ('a, 'b) t
val first :
  ('a, 'b) t -> ('a * 'c, 'b * 'c) t
val second :
  ('a, 'b) t -> ('c * 'a, 'c * 'b) t
val dup : ('a, 'a * 'a) t
val fst : ('a * 'a, 'b) t
val snd : ('a * 'b, 'b) t
```

Figure 1. Classic arrow interface

```
type 'a t
val map : ('a -> 'b) -> 'a t -> 'b t
val pure : 'a -> 'a t
val both : 'a t -> 'b t -> ('a * 'b) t
val fst : ('a * 'b) t -> 'a t
val snd : ('a * 'b) t -> 'b t
```

Figure 2. Applicative interface

`fst`, `snd`, `dup` and `second` operations in the interfaces. These make the dependency structure of the computation graph visible, allowing for more efficient implementations.

One way to think about the difference between arrows and applicatives is that applicatives represent computation trees, whereas arrows can represent more general (acyclic) computation graphs. Fig.3a shows a simple computation graph and how to build it with the arrow interface. Fig.3b shows a similar computation tree and how to build it with the applicative interface. This tree is the closest the applicative interface can get to representing that graph: it cannot represent that the same `map f` node should be used as input to both other nodes.

$$\begin{tikzcd}
& {} \arrow[d] & \\
& f \arrow[dl] \arrow[dr] & \\
g \arrow[dr] & & h \arrow[dl] \\
& l \arrow[d] & \\
& {} &
\end{tikzcd}$$

```
arr f
>>> dup
>>> first (arr g)
>>> second (arr h)
>>> arr l
```

(a) Arrow term

$$\begin{tikzcd}
{} \arrow[d] & & {} \arrow[d] \\
f \arrow[d] & & f \arrow[d] \\
g \arrow[dr] & & h \arrow[dl] \\
& l \arrow[d] & \\
& {} &
\end{tikzcd}$$

```
fun x ->
  let a = map f x in
  let b = map g a in
  let c = map h a in
  map l (both b c)
```

(b) Applicative term

Using the arrow interface, a graph with inputs of types a_1 … a_n and output of type `b` is represented by a value of type `(`a_1 `*` … `*` a_n`, b) t`. Using the applicative interface, a similar tree is represented by a value of type a_1 `t *` … `*` a_n `t -> b t`. Using the ordinary function type makes the applicative interface much easier to use, but prevents it from being able to encode the *sharing* necessary to represent a graph.

[1]This is the `Monoidal` variation on the applicative interface from Patterson and McBride[1, Section 7].

```
module Builder : sig
  type ('a, 'k) t
  val return : 'a -> ('a, 'k) t
  val bind :
    ('a, 'k) t
    -> ('a -> ('b, 'k) t)
    -> ('b, 'k) t
end
module Node : sig
  type ('a, 'k) t
  val pure : 'a -> ('a, 'k) t
  val both :
    ('a, 'k) t -> ('b, 'k) t
    -> ('a * 'b, 'k) t
  val fst : ('a * 'b, 'k) t -> ('a, 'k) t
  val snd : ('a * 'b, 'k) t -> ('b, 'k) t
  val map :
    ('a -> 'b) -> ('a, 'k) t
    -> (('b, 'k) t, 'k) Builder.t
end
```

Figure 4. Proposed interface (threaded)

2 Heaps also describe sharing

Managing sharing through manual application of the structural rules is a natural consequence of trying to represent sharing in an algebraic structure. We can instead use a non-algebraic approach. Heaps with dynamic allocation of mutable state are fundamentally about representing sharing. For instance, in Haskell we can write following term in the ST monad, to produces a value whose underlying structure is the same as that of our arrow computation:

```
a <- newSTRef x
b <- newSTRef a
c <- newSTRef a
newSTRef (b, c)
```

In particular, the equality of `STRef` will tell you that the `STRef`s contained in `b` and `c` are the same.

Inspired by this, we can define an interface for our computation graphs that treats nodes in the graph like pieces of mutable state. Fig.4 shows the interface.

There is a monad, `Builder.t`, similar to the `ST` monad, which represents computations that build graphs. We call its `'k` type parameter the *thread*. There is a type, `Node.t`, that represents nodes within the computation graph. It's interface is the same as the classic applicative interface, except that `map` is an operation of the `Builder.t` monad rather than an ordinary function.

Note that `Builder.t` should obey the monad laws, and `Node.t` should obey the applicative laws suitably adjusted.

3 Translating between the interfaces

We have implemented a translation to and from the proposed interface as a pair of OCaml functors[4]. Given an instance of our proposed interface, it implements the classic arrow interface for the type:

```
type ('a, 'b) prog =
 { prog : 'k. ('a, 'k) Node.t
     -> (('b, 'k) Node.t, 'k) Builder.t }
```

Given a traditional arrow, it produces an instance of our new interface with abstract builder and node types, along with total conversion functions between the original arrow type and the `prog` type.

The implementation of the new interface is done using an extensible variant type for the graph nodes and with a heterogeneous map as the heap. These translations form an isomorphism up to the various arrow, monad and applicative laws.

Round-tripping through our translations does not construct any additional `arr` nodes. It uses the builtin `fst`, `snd` and `dup` nodes instead. This ensures that any dependency structure visible to the arrow implementation before translation is still visible after translation.

4 Ignoring state threads

The interface is noticeably complicated by the existence of the thread type parameter. However, using a node with the wrong thread is not a very common bug for users to write and can be easily detected at graph construction time. Thus a reasonable case can be made for simply dropping the threads, giving the interface shown in Fig.5.

```
module Builder : sig
  type 'a t
  val return : 'a -> 'a t
  val bind :
    'a t -> ('a -> 'b t) -> 'b t
end
module Node : sig
  type 'a t
  val pure : 'a -> 'a t
  val both : 'a t * 'b t -> 'a * 'b t
  val fst : ('a * 'b) t -> 'a t
  val snd : ('a * 'b) t -> 'b t
  val map :
    ('a -> 'b) -> 'a t -> 'b t Builder.t
end
```

Figure 5. Proposed interface (threadless)

5 Related work

Bernardy and Spiwack[5] propose using a translation from linear functions to symmetric monoidal categories as an alternative to arrows. It achieves similar aims to our proposal, but involves quite a lot of heavy machinery in comparison.

There are a number of existing OCaml libraries that are implicitly using the interface we propose (e.g. Incremental[6]). These libraries present themselves as providing an applicative interface, however they actually use effectful `map` functions to detect sharing and produce a computation graph that is not a tree. They are essentially using the ambient monad of OCaml to fill the role of the `Builder.t` monad in our proposed interface.

References

[1] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[2] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1-3):67–111, 2000.

[3] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic notes in theoretical computer science*, 229(5):97–117, 2011.

[4] Split arrow library. https://github.com/lpw25/split-arrow/.

[5] Jean-Philippe Bernardy and Arnaud Spiwack. Evaluating linear functions to symmetric monoidal categories. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, pages 14–26, 2021.

[6] Yaron Minsky. Seven implementations of incremental. https://blog.janestreet.com/seven-implementations-of-incremental/, 2016.