StackExchange ▾ sign up log in tour help ▾

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

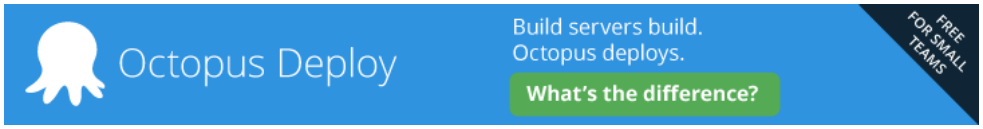
Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

Haskell: How is <*> pronounced?



Sorry, I don't really know my math, so I'm curious how to pronounce the functions in the Applicative typeclass:

```
(<*>) :: f (a -> b) -> f a -> f b
(*>)  :: f a -> f b -> f b
(<*)  :: f a -> f b -> f a
```

(That is, if they weren't operators, what might they be called?)

As a side note, if you could rename `pure` to something more friendly to podunks like me, what would you call it?

haskell operators pronunciation

asked Jul 13 '10 at 23:33

J Cooper
8,641 ● 4 ● 46 ● 81

5 @J Cooper... would you be able hear how we pronounce it? :) You might want to post a request on meta.stackoverflow.com for a voice recording and playback feature :). – Lirik Jul 13 '10 at 23:39

7 It's pronounced "Good grief, they were really running out of operators, weren't they?" Also, a good name for `pure` might be `makeApplicative`. – Chuck Jul 13 '10 at 23:44

2 @Lirik foh-net-iks – Tyler Jul 13 '10 at 23:45

4 (<*>) is the Control.Applicative version of Control.Monad's "ap", so "ap" is probably the most appropriate name. – Edward KMETT Jul 15 '10 at 5:20

8 i'd call it a cyclops, but that's just me. – RCIX Aug 6 '10 at 22:40

3 Answers

> Sorry, I don't really know my math, so I'm curious how to pronounce the functions in the Applicative typeclass

Knowing your math, or not, is largely irrelevant here, I think. As you're probably aware, Haskell borrows a few bits of terminology from various fields of abstract math, most notably Category Theory, from whence we get functors and monads. The use of these terms in Haskell diverges somewhat from the formal mathematical definitions, but they're usually close enough to be good descriptive terms anyway.

The `Applicative` type class sits somewhere between `Functor` and `Monad`, so one would expect it to have a similar mathematical basis. The documentation for the `Control.Applicative` module begins with:

> This module describes a structure intermediate between a functor and a monad: it provides pure expressions and sequencing, but no binding. (Technically, a strong lax monoidal functor.)

Hmm.

```
class (Functor f) => StrongLaxMonoidalFunctor f where
    . . .
```

Not quite as catchy as `Monad`, I think.

What all this basically boils down to is that `Applicative` doesn't correspond to any concept that's

particularly *interesting* mathematically, so there's no ready-made terms lying around that capture the way it's used in Haskell. So, set the math aside for now.

If we want to know what to call `(<*>)` it might help to know what it basically means.

So what's up with `Applicative`, anyway, and why *do* we call it that?

What `Applicative` amounts to in practice is a way to lift *arbitrary* functions into a `Functor`. Consider the combination of `Maybe` (arguably the simplest non-trivial `Functor`) and `Bool` (likewise the simplest non-trivial data type).

```
maybeNot :: Maybe Bool -> Maybe Bool
maybeNot = fmap not
```

The function `fmap` lets us lift `not` from working on `Bool` to working on `Maybe Bool`. But what if we want to lift `(&&)`?

```
maybeAnd' :: Maybe Bool -> Maybe (Bool -> Bool)
maybeAnd' = fmap (&&)
```

Well, that's not what we want *at all*! In fact, it's pretty much useless. We can try to be clever and sneak another `Bool` into `Maybe` through the back...

```
maybeAnd'' :: Maybe Bool -> Bool -> Maybe Bool
maybeAnd'' x y = fmap ($ y) (fmap (&&) x)
```

...but that's no good. For one thing, it's wrong. For another thing, it's *ugly*. We could keep trying, but it turns out that there's *no way to lift a function of multiple arguments to work on an arbitrary `Functor`*. Annoying!

On the other hand, we could do it easily if we used `Maybe`'s `Monad` instance:

```
maybeAnd :: Maybe Bool -> Maybe Bool -> Maybe Bool
maybeAnd x y = do x' <- x
                  y' <- y
                  return (x' && y')
```

Now, that's a lot of hassle just to translate a simple function--which is why `Control.Monad` provides a function to do it automatically, `liftM2`. The 2 in its name refers to the fact that it works on functions of exactly two arguments; similar functions exist for 3, 4, and 5 argument functions. These functions are *better*, but not perfect, and specifying the number of arguments is ugly and clumsy.

Which brings us to the paper that introduced the Applicative type class. In it, the authors make essentially two observations:

- Lifting multi-argument functions into a `Functor` is a very natural thing to do
- Doing so doesn't require the full capabilities of a `Monad`

Normal function application is written by simple juxtaposition of terms, so to make "lifted application" as simple and natural as possible, the paper introduces *infix operators to stand in for application, lifted into the `Functor`*, and a type class to provide what's needed for that.

All of which brings us to the following point: **`(<*>)` simply represents function application--so why pronounce it any differently than you do the whitespace "juxtaposition operator"?**

But if that's not very satisfying, we can observe that the `Control.Monad` module also provides a function that does the same thing for monads:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

Where `ap` is, of course, short for "apply". Since any `Monad` can be `Applicative`, and `ap` needs only the subset of features present in the latter, we can perhaps say that **if `(<*>)` weren't an operator, it should be called `ap`.**

We can also approach things from the other direction. The `Functor` lifting operation is called `fmap` because it's a generalization of the `map` operation on lists. What sort of function on lists would work like `(<*>)`? There's what `ap` does on lists, of course, but that's not particularly useful on its own.

In fact, there's a perhaps more natural interpretation for lists. What comes to mind when you look at the following type signature?

```
listApply :: [a -> b] -> [a] -> [b]
```

There's something just so tempting about the idea of lining the lists up in parallel, applying each function in the first to the corresponding element of the second. Unfortunately for our old friend `Monad`, this simple operation *violates the monad laws* if the lists are of different lengths. But it makes a fine `Applicative`, in which case `(<*>)` becomes a way of **stringing together a generalized version of `zipWith`, so perhaps we can imagine calling it `fzipWith`?**

This zipping idea actually brings us full circle. Recall that math stuff earlier, about monoidal functors? As the name suggests, these are a way of combining the structure of monoids and functors, both of which are familiar Haskell type classes:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

What would these look like if you put them in a box together and shook it up a bit? From `Functor` we'll keep the idea of a *structure independent of its type parameter*, and from `Monoid` we'll keep the overall form of the functions:

```
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ?
    mfAppend :: f ? -> f ? -> f ?
```

We don't want to assume that there's a way to create an truly "empty" `Functor`, and we can't conjure up a value of an arbitrary type, so we'll fix the type of `mfEmpty` as `f ()`.

We also don't want to force `mfAppend` to need a consistent type parameter, so now we have this:

```
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ()
    mfAppend :: f a -> f b -> f ?
```

What's the result type for `mfAppend`? We have two arbitrary types we know nothing about, so we don't have many options. The most sensible thing is to just keep both:

```
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ()
    mfAppend :: f a -> f b -> f (a, b)
```

At which point `mfAppend` is now clearly a generalized version of `zip` on lists, and we can reconstruct `Applicative` easily:

```
mfPure x = fmap (\() -> x) mfEmpty
mfApply f x = fmap (\(f, x) -> f x) (mfAppend f x)
```

This also shows us that `pure` is related to the identity element of a `Monoid`, so other good names for it might be anything suggesting a unit value, a null operation, or such.

That was lengthy, so to summarize:

- **`(<*>)` is just a modified function application, so you can either read it as "ap" or "apply", or elide it entirely the way you would normal function application.**
- **`(<*>)` also roughly generalizes `zipWith` on lists, so you can read it as "zip functors with", similarly to reading `fmap` as "map a functor with".**

The first is closer to the intent of the `Applicative` type class--as the name suggests--so that's what I recommend.

In fact, I encourage **liberal use, and non-pronunciation, of all lifted application operators**:

- `(<$>)`, which lifts a single-argument function into a `Functor`
- `(<*>)`, which chains a multi-argument function through an `Applicative`
- `(=<<)`, which binds a function that enters a `Monad` onto an existing computation

All three are, at heart, just regular function application, spiced up a little bit.

answered Jul 14 '10 at 1:55

C. A. McCann
63.7k ● 15 ● 163 ● 277

14 This is a fantastic answer. Extremely informative and very well-written. – Colin Cochrane Jul 14 '10 at 2:09

5 @Colin Cochrane: Are you sure you didn't misspell "long-winded" there? :) But hey, I'll take it! I always feel that `Applicative` and the functional idiomatic style it promotes don't get enough love, so I couldn't resist the chance to extol its virtues a bit as a means to explain how I (don't) pronounce `(<*>)`. – C. A. McCann Jul 14 '10 at 2:16

4 Would that Haskell had syntax sugar for `Applicative`'s! Something like `[| f a b c d |]` (as suggested by the original paper). Then we wouldn't need the `<*>` combinator and you would refer to such an expression as an example of "function application in a functorial context" – Tom Crockett Jan 6 '11 at 0:09

1 @FredOverflow: No, I meant `Monad`. Or `Functor` or `Monoid` or anything else that has a well-established term involving fewer than three adjectives. "Applicative" is merely an uninspiring, albeit reasonably descriptive, name slapped onto something that rather needed one. – C. A. McCann Sep 23 '11 at 18:30

1 @pelotom: see [stackoverflow.com/questions/12014524/… where kind people showed me two ways to get almost that notation. – AndrewC Aug 22 '12 at 18:04

Since I have no ambitions of improving on C. A. McCann's technical answer, I'll tackle the more fluffy one:

> If you could rename `pure` to something more friendly to podunks like me, what would you call it?

As an alternative, especially since there is no end to the constant angst-and-betrayal-filled cried against the `Monad` version, called "`return`", I propose another name, which suggests its function in a way that can satisfy the most imperative of imperative programmers, and the most functional of...well, hopefully, everyone can complain the same about: `inject`.

Take a value. "Inject" it into the `Functor`, `Applicative`, `Monad`, or what-have-you. I vote for "`inject`", and I approved this message.

edited Sep 3 '14 at 13:20

TRiG
4,866 ● 2 ● 30 ● 68

answered Jul 14 '10 at 22:15

BMeph
1,047 ● 8 ● 12

3 I usually lean toward something like "unit" or "lift", but those already have too many other meanings in Haskell. `inject` is an excellent name and probably better than mine, though as a minor side note, "inject" is used in--I think--Smalltalk and Ruby for a left-fold method of some sort. I never understood that choice of name, though... – C. A. McCann Jul 14 '10 at 23:45

3 This is a very old thread, but I think that `inject` in Ruby & Smalltalk is used because it is like you are "injecting" an operator between each element in the list. At least, that's how I always thought of it. – Jonathan Sterling Jun 7 '12 at 18:48

1 To *again* pick up that old side-thread: You're not injecting operators, you're replacing (eliminating) constructors that are already there. (Viewed the other way round, you're *injecting* old data into a new type.) For lists, elimination is just `foldr`. (You replace `(:)` and `[]`, where `(:)` takes 2 args and `[]` is a constant, hence `foldr (+) 0 (1:2:3:[])` ⇝ `1+2+3+0`.) On `Bool` it's just `if - then - else` (two constants, pick one) and for `Maybe` it's called `maybe` … Haskell has no single name/function for this, as all have different types (in general elim is just recursion/induction) – nobody Mar 16 '13 at 3:56

I always liked `wrap`. Take a value and wrap it in a Functor, Applicative, Monad. It also works well when used in a sentence with concrete instances: `[]`, `Maybe`, etc. "It takes a value and wraps it in a `x`".

answered Apr 18 '15 at 15:48

Peter Hall
4,162 ● 1 ● 17 ● 50