Built-in Mapping for More Powerful, Intuitive Code

Marshall Lochbaum

2/04/2012

1 Introduction

Let's say I give you a box, and tell you to eat. What are you going to do? Do you shout back at me that the task is impossible because it's a box? No! You open the box to reveal delicious fried chicken and biscuits, and indulge.

Now let's consider the same situation in programming. I give my computer a list of integers, tell it to multiply by two, and it spits back at me some red letters saying that it can't multiply a list and a scalar. Well, not always: there's a language called J which will do exactly what you meant in this case. Here's a sample:

```
   2 * 1 2 3
2 4 6
```

The space-separated numbers are—you guessed it—lists. In this paper, we'll explore a more general theory of "boxes," which replicates J's powerful application techniques and extends them to a much greater variety of data structures. We'll explore how this can shorten and clarify code, and how it allows a programmer to write very general code without having to worry about compound data types.

2 Notation

I'd like to focus on the ideas here, so I'll keep the notation as simple as possible. We're going to be writing expressions in lisp syntax—code is written like

```
(f x0 x1 (g y0 y1 y2) x3)
```

Where `f` and `g` are functions, and `x0`, etc. are their arguments. The code will be indented and its output won't be. I actually have an interpreter for this language that I plug all the examples here into, so the code and outputs that appear here are the honest truth.

This paper is all about data structures, so we'll need quite a few of them. Each one is represented as a function with some arguments.

- To start off, a list is something like `(, 1 2 3)`, or possibly `(, , S R ; N L)`, for a list of the functions which make data structures. Although the topics here apply equally well to homogeneous arrays, we'll use lists with mixed-type elements.
- A set is constructed with `S`; sets are unordered and don't care about multiples of items. Thus a set `(S Hi , Everybody !)` is the same as `(S , Everybody Hi ! ! !)`, and both are valid inputs. We'll output sets with all elements unique and sorted, but that's not the only way to do it.
- A reference is represented by `(R x)`. There's only one x, but the reference can also be null, written as `N`.

- We'll also make use of linked lists, which we can build with the other data structures, like so: `(R (, 0 (R (, 2 (R (, 5 N))))))`. That's a bit hard on the eyes. How about `;`, which stands for `R` composed with `,`? Then we get `(; 0 (; 2 (; 5 N)))`. Well, I'm still not too happy with this notation. Occasionally I'll give up and write `(L 0 2 5)` for a linked list.
- We can also construct trees using `;`, in the obvious way: `(; 2 (; 5 6) 6 (; 8 9))`. I'll let you work out binary trees as a particularly optional exercise.
- An associative array maps keys to values, so it stores each possible key along with the corresponding value. There's basically no way to write this in lisp that isn't awkward. I'll opt for `(M (, k0 v0) (, k1 v1) (, k2 v2))`, where the `M` stands for map. Since "map" is being used rather heavily in this paper, I'll call these "associative arrays" rather than maps.
- An error is just a string that says it's an error, like so: `(E "You left out a parenthesis, you dolt!")`. Never forget the parentheses.

3 Thinking inside the box

So how does the metaphor of a box extend to programming? I hope you've figured out what I mean to some extent, but there's a lot more to it—stay tuned! Starting at the basics, there's a particular way of looking at these data structures, and in fact most data structures, that I'd like to consider. In Haskell it's called a Functor. That's a word lifted from category theory and then used incorrectly, so I'll do without it in this paper. I'll call them containers, even though that term also shows up in programming and category theory, where it means something a bit weaker than what I want to convey. Actually, I'll just call them "values" most of the time, since every object used in this text can be mapped over.

A container contains things, in a very abstract sense. Usually there's a way to get the things out, but that's kind of tangential to this paper. Instead, we're going to try to make use of them as they are in some sense. Here's where the "box" metaphor starts to break down: We can't always just open the box. For example, an array contains elements, but if we tried to just "open" the array to get the elements out, we wouldn't have anywhere to put them—we'd need another array. So what we'll do is try to get functions to act inside our containers.

This leads to our working definition of a container: given a function, we can "map" the function over the container to get another container. Now the big idea is that our language should perform this mapping automatically. The following are a few examples of mapping in action.

3.1 Lists and collections

Well, it's pretty obvious how to map over lists: most programming languages define a function `map` that does just that. So, in our programming language we have (where `sq` is the function that squares its argument)

```
  (sq (, 1 2 3))
(, 1 4 9)
```

It's similarly easy to map over other types of collections. For example, a set will behave the same way as a list, as shown above. However, note that the implementation of the mapping is completely different: Our set will have to check that the resulting elements are unique and reorder them so it's in a correct format.

```
  (sq (S 1 2 3))
(S 1 4 9)
```

3.2 References

This one is also easy: dereference the pointer, apply the function, and take a pointer to the result. However, there's one little catch: the pointer might be null. In this case we just return another null pointer. Let's see it at work on some sample arguments:

```
  (sq (, (R 5) (R 6) N (R 2)))
(, (R 25) (R 36) N (R 4))
```

Yes, of course I just mapped that over a list. How else am I going to do multiple examples?

Now we also have the ingredients to map over linked lists, which if you will recall are just built from references and two-element lists. Mapping over an empty list (the same as `N`) will give us another empty list, while mapping over a (value, successor) pair applies the function to the value, and then to the rest of the list, the successor. Sounds quite a bit like the recursive definition of mapping, and it works exactly the same as the other collections.

3.3 Functions

Wait, what? A function isn't a container, right? Well, it is. The value that it "wraps" is its return value. You actually do this all the time: think of all the times you've written "sin" to mean "$\sin(x)$." Though it may have struck you as a violation of mathematical notation at the time, it's actually a more reliable convention. What if the variable wasn't x but θ? With function mapping, you manipulate functions, not arguments, and you just have to remember to apply it to something once you've built up the correct function.

Our rule here is that we map a function over another function by composing the two: $f(g)$ gives $f \circ g$. The argument function "contains" a value that we get out by applying it, and the returned function "contains" another value in the same way. When we map over the function, we get something my interpreter calls a "fork," for reasons the J programmers in the audience are giggling about right now.

```
  (sq -)
(fork sq -)

  ((sq -) (, 2 4 5) (, 3 2 1))
(, 1 4 16)

  ((sum (sq -)) (, 2 4 5) (, 3 2 1))
21
```

And there you have the square of the distance—about as simple as saying "the sum of the squares of the difference."

One incredible property of this interpretation of mapping for functions is that it makes function application associative. We could just as easily have written `((sum sq) -)` as `(sum (sq -))` in the above expression. The first evaluates `(sum sq)` to give us a composition, then calls that on `-`, which will call `sq` and then `sum` on `-`, or exactly `(sum (sq -))`. Perhaps a less roundabout way to say this is that mapping changes application to composition, which is associative.

3.4 Associative arrays

Associative arrays are essentially another implementation of functions; we just enumerate the domain and range. Therefore, they map like normal functions. All we have to do is apply the mapped function to each of the values:

```
  (sq (M (, 'energy' 3) (, 'mass' 4)))
(M (, 'energy' 9) (, 'mass' 16))
```

That's pretty good, as long as you don't forget to relabel.

3.5 Errors

When a piece of code results in an error, the error should be visible pretty much immediately. One thing we don't want it to do is to disappear or mutate. Thus mapping over an error should give the same error, without even bothering to apply the function.

```
  (/ 1 0)
Error: Division by zero
  (sq (/ 1 0))
Error: Division by zero
```

That way, we get the errors out of a program when they happen, rather than errors telling us that we tried to square an error. This is a necessity if a language is going to manipulate errors as values, which is a particularly powerful paradigm because it makes errors first-class objects. That prevents the programmer from being shackled by difficult-to-manipulate error values.

4 Domains

So when do we pull out the map to direct our functions? When they're lost, of course! When a function knows how to handle an argument, we don't need to map over the argument. On the other hand, if I'm little old square function and you tell me to get to work on an entire array, I'm going to need some divine inspiration to get the job done. So, I point out that I have no clue what I'm supposed to be doing, and the interpreter steps in, sees the argument, and tells me, "just go one at a time."

There's a pretty straightforward way to do this. Each function has a domain that it "knows" how to handle. Then when it gets applied to something outside of its domain, it maps over it. If you look back at the places where we mapped functions and those where we didn't, it's clear what the domain of a function like square is.

The idea of mapping when a function needs to map, rather than when a user tells it to map, makes the mapping much more flexible and powerful. If I made a tree structure using lists, it would be difficult to instruct a function how to map over it. Also, I could easily make the mistake of thinking a single "map" command should map the function over the entire tree, and end up with lots of errors when it only works for one level.

5 Multiple arguments

Okay. Now add `(, 4 7)` and `(, 3 5 4)`. You can't? Don't give me that nonsense. I already told you how mapping worked.

It's true—regardless of how hard you try, there's nothing in the above telling you how to do something simple like add two lists. But adding two lists is something really natural. In fact, you probably didn't notice when I slipped into the above

```
  ((sq -) (, 2 4 5) (, 3 2 1))
(, 1 4 16)
```

I know I didn't. I just noticed this now, as I am writing this paragraph, a page later. So what happens, exactly?

```
  (- (, 3 2 1) (, 2 1 0))
(, 1 1 1)
```

This is completely unsurprising. I bet it's the same result you would get if you handed the problem to your average third grader (with, you know, less parentheses and more nice tables). Now how do we accomplish it?

In addition to the basic map I showed above, we need another map, to be defined on the same data structures but with multiple arguments. That's our criterion: given a function and multiple arguments, we can map the function over all of those arguments. This also will give us the map for one argument. It's important that the multiple-argument and single-argument maps be compatible, but it turns out that there's a lot more flexibility with maps on multiple arguments. All the collections act the same under simple mapping, but when we map over multiple arguments they'll act differently. Let's see how!

5.1 Functions

A number of the data structures above can be thought of as functions with particular domains. We have lists, which map the indices $(0, 1, 2 \ldots)$ to values, associative arrays, which map their keys to values, and functions, which map other things to values. Then all we have to do to map over is make a new function which takes any number of functions, applies them all, and acts on them. For arrays this looks like

```
  (* (, 3 4 2) (, 2000 1500 4000))
(, 6000 6000 8000)
```

With numbers we get another `fork`. Applying that gives us values.

```
  (* + -)
(fork * + -)
  ((* + -) (, 5 25) (, 4 24))
(, 9 49)
```

That's a simple bit of code for the factored difference of squares—you can read it as "the product of the sum and the difference." In fact, the whole thing translates to the remarkably english-like "the product of the sum and the difference of the list of five and twenty-five and the list of four and twenty-four," by a very naive translation.

Associative arrays work similarly. If we have two arrays for the time and space taken by programs, we can multiply them together to get a reasonable measure of overall performance:

```
  (*
    (M (, 'bogosort' 3000000) (, 'bubblesort' 100) (, 'quicksort' 40))
    (M (, 'bogosort' 11)      (, 'bubblesort' 12)  (, 'quicksort' 20)))
(M (, 'bogosort' 33000000) (, 'bubblesort' 1200) (, 'quicksort' 800))
```

There is a question, though, of what to do with domains. If the two functions have different domains, then the domain of the result has to be the intersection of those domains. We can think of it this way: if an input is outside of the domain, then applying (or indexing into) the function (or array) will give us back a domain error or an index error or a key error. Then, when we apply the function to those results, it returns an error because it maps over errors. Therefore if we apply the function to a value that is outside the domain of any of the arguments we mapped over, it will give us an error, and we were outside of the domain. Here's what we get for two partial associative arrays:

```
  (+
    (M (, 'a' 12) (, 'b' 3) (, 'd' 8))
    (M (, 'b' 5) (, 'c' 1000) (, 'd' 2)))
(M (, 'b' 8) (, 'd' 10))
```

5.2 References, linked lists, and trees

The rule for references is simple: if any reference is null, we don't have enough information to apply the function, so we return null. Otherwise, we dereference everything, apply the function, and take a reference to that.

```
   (+ (R 3) N)
N
   (+ (R 3) (R 5))
(R 8)
```

This immediately gives us the rule for linked lists. If any of the linked lists is empty, we stop there, and if not, we apply the function on the first element of each, and append that to the function applied to the rest. Looks like it came straight from an introduction to recursion in programming. Basically, it will truncate to the length of the shortest list, and map as you would expect. Trees are the same, but with more, uh, branching.

5.3 Sets

Sets act very differently from the other data structures in this discussion. They don't have any ordering, so we can't just match up elements exactly. In fact, we have only one well-defined option for a mapping that is consistent with the single-argument mapping. That's to take the product of all the sets (The set of all the ordered tuples with one element from each set), and apply the function to each element of that. Here's the product of a few sets:

```
   (map , (S 1 2 3) (S - +) (S 'Hello' 'World'))
(S (, 1 + 'Hello') (, 1 + 'World') (, 1 - 'Hello') (, 1 - 'World')
   (, 2 + 'Hello') (, 2 + 'World') (, 2 - 'Hello') (, 2 - 'World')
   (, 3 + 'Hello') (, 3 + 'World') (, 3 - 'Hello') (, 3 - 'World'))
```

All functions are in the domain of `,` (it's very reasonable to take a list of three sets), so I had to map it over the arguments explicitly. You can see that the product gets rather big rather quickly. Now if we replace the `,` with another function, we get that same set above where the function is applied to each tuple. Let's find all the sums of two positive squares which are smaller than 20:

```
   (+ (S 1 4 9 16) (S 1 4 9 16))
(S 2 5 8 10 13 17 18 20 25 32)
```

This rapidly becomes handy in combinatorial problems. It also precisely matches the notation that group theorists use when they talk about sets—you'll commonly see $H + G$ taken to mean exactly what it does above.

6 More domains, and inhomogeneous arguments

You can see from the comments on domains above, which I've intentionally kept vague, that they work in essentially the same way as the single-argument case. In fact, that's only true for the case when everything maps together smoothly and simply. Adding a tree and a set, for example, is just impossible.

The solution to that problem is a simple one, and boils down to defining a number of "mapping classes" each of which describes how a particular data type maps. Then compatible data types have the same mapping class, and incompatible ones don't. It's all rather straightforward, so I'll allow you to ponder the details on your own time. If you're really bored.

However, I have one last surprise for you, which will round out this introduction and make the system we have developed fantastically powerful and flexible. As if it weren't already.

Let's go back to the J code from the introduction:

```
   2 * 1 2 3
2 4 6
```

I promised to replicate J's behavior, and I still haven't addressed this example! Nowhere in the text do I tell you how to multiply a list by a scalar! Well, we need more functionality. There are actually two very good ways to think about this one. Here's one solution: Each data type has another property associated with it:

it can embed any value into that data type. If I want to make 1 into a function, I get the constant function that returns 1. If I want a set, I get `(S 1)`. For a reference, `(R 1)`. After that we can map normally. Note, however, that this isn't always a great solution. If I try to wrap 1 into an associative array, I will get out the array that associates every possible key with 1. That probably won't go over too well with the people who are running their computations with a finite amount of RAM.

The second interpretation is to say that we "bind" the `2` to `+`, and then map over the rest of the arguments—here there's only one—with the new function. We still map over the appropriate arguments, but the arguments we don't have to map over are used, unmodified, for every single invocation of the function. I think this one is a little less intuitive, but it's clear that it always works without the need to associate another function with our datatype. It's also a bit easier to implement, and it has better performance properties. However, there's one caveat: if we try to map in one type over a bunch of things of different types, we don't have any structure to use. We end up binding every one of our arguments to the function, and then trying to apply that to an empty list. There's just no way to map. That turns out not to matter, a fact which brings us back to types...

How do I choose which type's version of "map" to use? Back to domains. When I apply a function, some arguments will be in the domain, and some will be out of it. Now I look at the arguments that are outside of the domain—those are the ones I want to map over. If they have mixed types, then I can't do it, and I return an error telling my programmer that he's thinking hazily (and I suppose I couldn't go wrong with specifics on where he misstepped). Otherwise I take shared type of all the arguments, and map with that type. Now there has to be at least one argument with the mapping type, because otherwise everything would be in the domain and I wouldn't map in the first place. That means I can safely apply either of the methods above.

This careful treatment of domains and types means that I can say things like `(+ 1 sq)` without qualms. I can multiply vectors by scalars, and even do something complicated like a matrix product:

```
  ((sum *)  (, (, 1 0) (, -1 1))  (, 5 3))
(, 2 3)
```

The matrix is two column vectors, and we multiply it by a column vector, so this is the same as $\left[\begin{smallmatrix}1 & -1\\ 0 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}5\\ 3\end{smallmatrix}\right] = \left[\begin{smallmatrix}2\\ 3\end{smallmatrix}\right]$.

6.1 What about lists?

There are actually two, equally valid, ways to think about mapping over lists. Both of them are the same for the one-argument case, and consistent for the multiple-argument case.

The first, the "array" way of thinking, says that you can't map over two lists of different length. `(, 2 4 1)` and `(, 8 4)` can't even inhabit the same vector space! This means that there is a different "mapping class" for each length of list. Then to make, say, `3` into a four-element list, I take `(, 3 3 3 3)`.

The second, which I'll call the "linked-list" way of thinking, says that you can map over multiple lists, simply by truncating the longer ones. Then all lists form one mapping class, and to make `3` into a list we take the infinite list of `3`'s.

I've taken the slightly diplomatic approach of using regular lists as arrays, and reserving the second for linked lists, where it arises naturally as a consequence of the way references are handled.

7 Type theory

I'm finally through with all my arguments here. However, I'd like to share with those who breathe in type constructors a formal definition of the methods illustrated above. Everything is written in Haskell, which I've made very little attempt to clarify for the uninitiated. (However, I haven't used much Haskell and you can certainly study the language a little and understand what I've said in this section).

This approach is incompatible with Haskell types as it involves polymorphism at a level much too complex for a static type system. Thus we make everything of type `Item`. Our ultimate goal is to attain a function

```
apply :: Item -> [Item] -> Item
```

which will replicate the application described in this paper.

A map is derived from the Haskell concept of a `Functor`, which is a type constructor `f` with the method

```
fmap :: (a->b) -> f a -> f b
```

We need to make a number of changes to this to get the required generality. First is to fix the types:

```
imap :: (Item->Item) -> (f Item)->(f Item)
```

But `f Item` should still be an `Item` (everything's an `Item`!), so we have

```
imap :: (Item->Item) -> (Item->Item)
```

Basically we want to make one application into another kind of application. But application isn't given by a function; it's given by an `Item`, via `apply`. So how about instead,

```
imap :: Item -> [Item] -> Item
```

One more thing: we need a domain to tell us how to map. So let's include one, as a list of `Bools`. We'll also find it useful to place the arguments to the function (the things we're mapping over) as the first arguments, since they're the ones that determine how to map.

```
imap :: [Item] -> [Bool] -> Item -> Item
```

Got it. Mapping takes a list of arguments, a list of which ones are in the domain, and finally a function to apply, and then applies it to produce one `Item`.

This is going to require a few more functions, to find which type to map over, and do the mapping with that type. For mapping types we use the type `MapClass`. Here they are:

```
mapClass :: Item->MapClass
getType :: [Item]->[Bool]->Maybe MapClass
_imap :: MapClass -> [Item]->[Bool]->Item->Item
```

So `getType` filters the `Items` by which are in the domain, then sends back a `Nothing` if they are not all the same and otherwise gives us `Just` the first one. Then `_imap` uses that to pick a particular mapping function, and applies it.

We also need to be able to take a function and figure out what's in the domain. Simple enough:

```
domain :: Item -> [Item] -> [Bool]
```

Oh, and we'd better be able to apply a function without mapping it, as well:

```
_apply :: Item -> [Item] -> Item
```

That's all the type signatures! For your convenience, they are listed together here:

```
apply :: Item -> [Item] -> Item

domain :: Item -> [Item] -> [Bool]

imap :: [Item] -> [Bool] -> Item -> Item

mapClass :: Item -> MapClass
getType :: [Item]->[Bool]->Maybe MapClass
_imap :: MapClass -> [Item]->[Bool]->Item->Item

_apply :: Item -> [Item] -> Item
```

`domain`, `mapClass`, `_imap`, and `_apply` are the hard ones—these depend on the function being applied and the mapping classes being applied to, and I won't discuss them here. `getType` is tedious. `apply` and `imap` are easy to implement, though, so I'll give the code for these:

```
apply f xs = let ds = domain f xs in
  if foldl (&&) True ds then _apply f xs else imap xs ds f

imap xs ds =
  case (getType xs ds) of Nothing -> Error "Incompatible mapping classes"
                          Just mc -> _imap mc xs ds
```

8 Acknowledgements and references

Very few of the ideas presented in this paper are actually new. My job has been to unify and extend existing paradigms, rather than create new ones. In particular, I've drawn from two languages, J and Haskell.

J is a language that is unique in its expressive power. In particular, the concepts of rank and forks, which are the restrictions of concepts listed here to arrays and functions, respectively, have provided the impetus for my consideration of mapping in general. The unique idea here is not the concept of mapping, but the idea that it's the language's job rather than the programmer's. The result is a very high degree of useful polymorphism: the same functions can work with scalar, vector, and matrix arguments without difficulty. I encourage anyone who hasn't seen J before to go to jsoftware.com and try it. I recommend as an introduction Henry Rich's *J for C programmers*, which is available on the J website and provides a very good description of rank.

Haskell is a somewhat more common language than J. Here, I've used concepts from the `Functor` and `Applicative` typeclasses. These provide a rigourous description of mapping, and extend it to multiple arguments. I was surprised to find while reading about them that `(->) a`—that is, a function with domain `a`—is a functor. The generality of the functor typeclass is reflected in the generality of the approach above. I used as an introduction to Haskell *Learn You a Haskell for Great Good!*, by Miran Lipovača, which has a more-than-adequate description of how functors, applicatives, and monads work.