Rebelsky - Higher-Order Graphics

Comments on:

Rebelsky, Samuel (2007). Higher-Order Graphics. Web resource available at `http://www.cs.grinnell.edu/~rebelsky/Courses/CS302/2007S/HOG/higher-order-graphics.html` (last modified 4 February 2007; visited 4 February 2007).

Comments

For the first time, I can see how powerful Scheme can be. Higher-Order programming allows the programmer to create ridiculously concise code.

Compose is good way to solve problem and thinking strategy (divide-and-conquer).

I'm not sure how composition supports divide-and-conquer.

Questions

This may just be from lack of experience with Higher-Order programming, but after doing the reading, it seems like the code could get very hard to follow. Do you think Higher-Order programming can be harder to read/understand and if so, do you think this is a problem?

Until you're used to thinking and reading "higher order", it is definitely quite difficult, in part because it is so concise. But it gets easier, just as reading recursive functions gets easier.

That noted, particularly when you're working in an interactive environment, some higher-order things are definitely "one off", and can therefore be "write only".

One of the things I gained from the reading is learning is how concise script-Fu can be. However, the more concise the code got, the harder it was for me to understand the code. Do you think it is easier to create complex images using script-Fu or using dialog boxes?

Well, someone has to write the program that implements the capacity of the dialog boxes, so that's a bit of an odd question. I think you are less constrained when you use Script-Fu, but if the drawing environment is well enough designed, you need not use Script-Fu. Of course, predicting everything that someone might want to do is impossible, so you'll always find some cases in which Script-Fu will let you do something that you can't do with dialog boxes.

Scheme daes not follow many of the design principles that the articles we have read suggest. For example: it is not strongly typed, many statements have hard to read syntax (e.g. if statements and lets), and there are parentheses all over the place. How is it, then, that so many people hail LISP and its close relatives as the ultimate language or why do the design principles not reflect LISP?

I'm not sure "parentheses all over the place" is something the authors we've read cautioned against. Hoare, in particular, asked for a simple, unambiguous syntax, which Scheme clearly has. We can debate how hard the syntax is to read. So, the big complaint you have is the type system, and that is a serious critique. So, why do people love LISP? They have different design criteria, and make different assumptions about programmers. We'll visit the alternative perspective next week and beyond.

I didn't really see the nested lambdas as simplifying or shortening code very much. Are there situations where using nested lambdas would be essential or extremely desirable or are nested lambdas always a small convenience?

Boy, it doesn't seem that they are every necessary. Often, the gain is in simplicity - you type less. It also turns out that defining some procedures with nested lambdas (also known as Currying*) makes it easier to partially apply functions.*

Perhaps this is because I don't have enough experience with functional programming, but Higher-Order programming doesn't seem all that different or more powerful then what can be achieved by nesting function calls in other function calls. While it is true that Higher-Order programming can represent code in fewer lines, but lines of code is a somewhat poor measurement.

```
if(i != 1)
  createList();

if (i != 1) {
  createList();
}
```

Both of these blocks of code are identical, but the second one is longer because it explicitly uses { and }. Perhaps a better illustration of what I'm thinking of is the Functional graphics presentation where they compared the line length of a set of transformations on primitive structures written in scheme and the same set of transformations written in C. The C code was more line because it needed to call functions like pushMatrix() and popMatrix() and because the transformations were done manually. If the transformations can simply been packaged in function calls, the difference between the number of lines for the code between the two languages wouldn't have been very great.

Yeah, sometimes we set up strawmen as targets. To me, the thing that really sold the functional approach has to do with refactoring (which we'll see a bit about in the next reading): Once you see a common control structure, you can make that control structure a higher-order function. You can't really do that in other languages. The rest I think of us being particularly concise or elegant, but it requires a lot more experience to be concise and elegant.

If your model is more interactive, it's also much nicer to be able to write short one-liners. Consider the problem of updating a vector of grades by adding 2. In Scheme, we can write

```
(map! (l-s + 2) grades)
```

In Java, the shortest thing we can write is

```
for (i = 0; i < a.length; i++) a[i] += 2;
```

It’s close in terms of size, but less so in terms of clarity because but some of the control is mixed. In the first, we can see that we’re doing something to each element in a vector, because we’re using `map!`. In the second, we have to look not just at the control structure, but also the body to figure out that we’re doing something to each element.

Since the process of currying can be considered to be a procedure "returning another procedure", is it a good functional programming technique? How much, if at all, does it make the program/procedure more efficient?

In languages in which everything is automatically Curried, it can help the programmer think about and expres things more concisely. It doesn’t have a big effect on program efficiency, though.

Can script-fu be used for image recognition? i.e., I am thinking of a case with a function that takes two parameters: image1 and image2. It then compares the two. I can imagine that it would not be too difficult to tell if two identical pictures are identical or not (just compare them pixel-by-pixel). How about a case where the pictures are not identical? This may require a fair bit of pattern recognition? Is there any robust image-recognition program written in script-fu?

No, I don’t think any such thing has been written in Script-Fu, although one of my students started to try to write something like that last summer.

One thing that seemed a bit contradictaory was that, on one hand, we are striving for concise code. On the other however, we have anonymous procedures, which seem to lengthen the code but are convenient. Is a convenient method better than a concise one?

It depends on your domain. In most cases, we strive for clarity above all. In Script-Fu, many things we write we write once and throw away. (E.g., “Hey, I need a red box that fits in this precise position.”) Programmer convenience wins there. However, I’ll also note that anonymous procedures do not really lengthen the code. If you need to name the procedure, you still have to write the lambda (which is the long part). As importantly, you can create anonymous procedures without using lambda (e.g., by using compose with existing procedures).

Are higher order procedures exclusive to a functional model? I don’t remember ever trying it, but would a function declaration like map(procedure, vector) work in C? What type would procedure be? How about Java?

In C, you can use function pointers as parameters to other procedures, which is one of the ways we can write generic sorting algorithms. I’ll admit that I always go back to K&R for the details. In Java, function pointers don’t exist, so you need to add your own Function object, which is generally not so concise.

Less-good Questions

I still haven’t seen much difference between script-fu and the scheme that we used in 151 apart from image-related functions.

You won't. It's just a dialect of Scheme.

As I read, at each (x,y), the image contains a particular color. What if different colors were assigned to the same (x,y). Would anything different occur or would the (x,y) just be the color of the last assignment?

It's just like an array (well, a two-dimensional array with only one kind of value): If you put a value at a particular location, you replace the old value.

I understand it's unnecessary to give a function a name sometime. But it won't hurt much to do so. Also, we never know the future, maybe we change our mind and want to reuse it. Are you worrying about name space?

It's the same principle as not requiring the programmer to name the temporaries that often must be created in an arithmetic expression - Don't make the programmer name things that don't need names. By avoiding names, we also get more concise code.

It appears that composition is not the most efficient way. e.g. compose square increment, we will visit each pixel twice. we could do better. However, O(2n)=O(n). so, it doesn't waste much.

I'm not sure what you mean by "we will visit each pixel twice", given that the example seems to be for a numeric composition, rather than a pixel-based one. And, while O(2n) = O(n), we do eventually worry about the constant multiplier. For example, something that takes 100n seconds is much less pleasant than something that takes n/1000 seconds.

I read and understood the article but found nothing that was particularly noteworthy or thought provoking.

Your goal is to strive to find such things, or at least such questions. Look at some of the questions above for ideas. It should always be possible to find an interesting question by putting together ideas from prevous readings with ideas from the current reading.