pissed off about functional programming

by mstone on Apr 24, 2005 at 11:46 UTC (replies)

http://perlmonks.org?node_id=450922

Okay.. fair warning: I'm venting.

I just had an long and **very** frustrating conversation with a young programmer who recently discovered functional programming, and thinks it can solve every problem in the world.

Before I go any farther, let me make one thing clear: **I do not hate functional programming.** On the contrary, I agree with every guru out there who says that you can't become a Real Programmer without learning functional programming. FP, in my never humble opinion, doth rock.

But I have to call bullshit on some of the misleading things that are commonly said about FP, because I've just had my head slammed against them in their most clueless and trivial form.

Myth #1 - Functional programming is Lambda Calculus

This is true for a specific value of 'true'.. the same one we use for the statement, 'computers do integer math'.

The limits of this truth become obvious when you try to store the value (2^N)+1 in an N-bit integer. The pedantically correct statement is, 'computers do integer math for operations whose values fall within a certain range'. We generally leave off the qualifier for convenience, but its importance can be summed up in three letters: **Y2K**. Or in terms of more recent events, 'Comair database'.

Functional programming approximates lambda calculus the way computers approximate integer math. It works just fine in the range where it's defined to work just fine, and blows chunks everywhere else. This is hardly a fatal limitation, but it does mean we should know where the limits are, and what they mean.

For functional programming, the limits mean you have to be aware of the simultaneity constraints inherent in lambda calculus, and the way they interact with the lazy evaluation techniques that are fundamentally necessary to implement FP in any kind of real-world computer.

Okay.. that's two pieces of vocabulary..

Simultaneity means that we assume a statement in lambda calculus is evaluated all at once. The trivial function:

```
    ^f(x) ::= x f(x)
```
[download]

defines an infinite sequence of whatever you plug in for x (I'm using the caret as a substitute for lambda because I'm not in a mood to mess with non-7-bit ASCII values at the moment). The stepwise expansion looks like this:

```
    0   - f(x)
    1   - x f(x)
    2   - x x f(x)
    3   - x x x f(x)
     ...
```
[download]

and so on.

The point is that we have to assume that the 'f()' and 'x' in step three million have the same meaning they did in step one.

At this point, those of you who know something about FP are muttering "referential transparency" under your collective breath. I know. I'll beat up on that in a minute. For now, just suspend your disbelief enough to admit that the constraint does exist, and the aardvark won't get hurt.

The problem with infinite expansions in a real-world computer is that.. well.. they're *infinite.* As in, "infinite loop" infinite. You can't evaluate every term of an infinite sequence before moving on to the next evaluation unless you're planning to take a **really** long coffee break while you wait for the answers.

Fortunately, theoretical logic comes to the rescue and tells us that preorder evaluation will always give us the same results as postorder evaluation.

More vocabulary.. need another function for this.. fortunately, it's a simple one:

```
    ^g(x) ::= x x
```
[download]

Now.. when we make the statement:

```
    g(f(x))
```
[download]

Preorder evaluation says we have to expand f(x) completely before plugging it into g(). But that takes forever, which is.. inconvenient. **Postorder** evaluation says we can do this:

```
    0   - g(f(x))
    1   - f(x) f(x)
    2   - x f(x) x f(x)
    3   - x x f(x) x x f(x)
     ...
```
[download]

which is a heck of a lot nicer.

The technique of **lazy evaluation** takes the theoretical correctness of postorder evaluation and applies it to the execution of code. The basic idea is that the language interpreter won't execute the code necessary to evaluate expansion step 300 until some part of the program actually calls for that value. This technique amortizes *quite* nicely, since it saves us, on average, an infinite amount of labor for each function.

Now.. with those definitions in place, you can see how the relationship between simultaneity and lazy evaluation isn't quite as simple as it appears at first glance. It's theoretically possible that the 'f()' or 'x' might change between the lazy evaluation of step one and the lazy evaluation of step three million. Trying to prevent that is what we programmers call a 'hard' problem.

Lambda calculus doesn't have this problem, the way integer math doesn't have register overflow issues. In lambda calculus, the infinite expansion of every function occurs instantly, and there's no way that any of the functions can possibly change because there's no time for them *to* change. Lazy evaluation brings time into the picture, and introduces order-of-execution issues. Trying to run functional logic in two or more simultaneous threads can raise serious problems, for instance.

In point of fact, the simultaneity constraints inherent in lambda calculus have been shown to make it unsuitable for issues involving concurrency (like multithreading), and for problems like that, it's better to use pi-calculus.

Myth #2 - Functional programming is 'different from' imperative programming.

There's some operator overloading in this issue, because 'imperative programming' has two different meanings.

On one hand, 'imperative' means 'a list of operations that produce a sequence of results'. We contrast that with 'declarative programming', which means 'a list of results which are calculated (somehow) each step of the way'. Mathematical proofs and SQL queries are declarative. Most of the things we think of as programming languages are 'imperative' by this meaning, though, including most functional languages. No matter how you slice it, Haskell is **not** declarative.

On the other hand, 'imperative' is also used as a contrast to 'functional', on the basis of really poorly defined terms. The most common version runs something like, "imperative languages use assignment, functional languages don't".. a concept I'll jump up and down upon later. But I'm getting backlogged, so we'll queue that one up and move on.

Getting back to the current myth, this is another one of those statements that's true for a given value of 'true'. In this case, it's the same value we use for the statement, "red M&Ms are 'different from' green M&Ms."

Church's thesis, upon which we base our whole definition of 'computing', explicitly states that Turing machines, lambda calculus, while programs, Post systems, mu-recursive grammars, blah-de-blah-de-blah, **are all equivalent.**

For those quick thinkers out there who've noticed that this puts Myth #2 in direct contradiction with Myth #1, ***KA-CHING!*** *you've just won our grand prize! Please contact one of the operators standing by to tell us how you'd like your sheepdog wrapped.*

Yes, the syntactic sugar of functional programming is somewhat different from the syntactic sugar of non-functional programming. Yes, functional programming does encourage a different set of techniques for solving problems. Yes, those techniques encourage different ways of thinking about problems and data.

No, functional programming is not some kind of magic pixie dust you can sprinkle on a computer and banish all problems associated with the business of programming. It's a mindset-altering philosophy and/or worldview, not a completely novel theory of computation. Accept that. Be happy about it. At least friggin' COPE.

Myth #3 - Functional programming is referentially transparent

This one is just flat-out wrong. It's also the myth that pisses me off the most, because the correct statement looks very similar and says something *incredibly* important about functional programming.

Referential transparency is a subtle concept, which is founded on two other concepts: **substitutability** and **frames of reference.**

More vocabulary..

Substitutability means you can replace a reference (like a variable) with its referent (the value) without breaking anything. It's sort of the opposite of removing magic numbers from your code, and is easier to demonstrate in violation than in action:

```
    my $x = 3;
    my $y = 3;

    if (3 == $x) {                  # substitutability works
        print "$x equals $y.\n";
    }

    $x++;

    if (3 == $y) {                  # substitutability fails
        print "$x equals $y.\n";
    }
```
[download]

Mutable storage (aka: assigning values to variables) offers an infinite variety of ways to make that kind of mistake, each more heavily obfuscated than the last. Assigning values to global variables in several different functions is a favorite. Embedding values (using $x to calculate some value that gets stored in $z, changing $x, and forgetting to update $z) is another. These problems are so common, and are such a bitch to deal with, that programmers have spent decades searching for ways to avoid them.

Frames of reference have a nice, clear definition in predicate calculus, but it doesn't carry over to programming. Instead, we have two equally good alternatives. On one hand, it can mean the scope of a variable. On the other hand, it can mean the scope of a value stored in a variable:

```
    my $x = 3;      # start frame of reference for variable $x
                    # start frame of reference for value $x==3

    my $y = 3;      # start frame of reference for variable $y
                    # start frame of reference for value $y==3
```

```
    if (3 == $x) {
        print "$x equals $y.\n";
    }

                    # end frame of reference for value $x==3
    $x++;           # start frame of reference for value $x==4

    if (3 == $y) {
        print "$x equals $y.\n";
    }

                    # end frame of reference for value $x==4
                    # end frame of reference for variable $x
                    # end frame of reference for value $y==3
                    # end frame of reference for variable $y
```

[download]

The notation above shows exactly why substitutabiliy fails in the second conditional. The frame of reference for the value $x==3 ends and a new one begins, but it happens implicitly, which means it's hard to see.

But there's another problem. If you look at that list of 'end' statements at the bottom, you'll notice that the frames of reference for $x and $y are improperly nested. The frame of reference for $x comes into existence before $y, and goes out of existence before $y. Granted, I did that specifically so I could talk about the problem, but you can do all sorts of obscene things to the nesting of value frames of reference with three or more variables.

Stepping back for a second, it's clear that substitutability is only guaranteed to work when all the values are in the same frame of reference as when they started. As soon as any value changes its frame of reference, though, all bets are off.

This is one of the biggest kludge-nightmares associated with mutable storage. Frames of reference for values pop in and out of existence, can be implicitly created or destroyed any time you add/delete/move/change a line of code, and get munged into relationships that defy rational description.

And don't even get me started on what conditionals do to them. It's like Schrodinger's cat on bad acid.

Functional programming changes that by declaring that the frame of reference for a variable and the frame of reference for its value **will always be the same.** The frame of reference for a value equals the

scope of its variable, which means that **every block is a well-defined frame of reference** and **substitutability is always guaranteed to work within a given block.**

This is what functional programming *really* has over mutable storage. But it isn't referential transparency.

Y'see, referential transparency is a property that applies to frames of reference, not to referents and references. To demonstrate this in terms of formal logic, let me define the following symbols:

```
    p1  ::= 'the morning star' equals 'the planet venus'
    p2  ::= 'the evening star' equals 'the planet venus'
    p3  ::= 'the morning star' equals 'the evening star'

    (+) ::= an operator that means 'two things which equal
            the same thing, equal each other'

    ⇒   ::= an operator which means 'this rule produces this result'
```
[download]

With those, we can generate the following statements:

```
    p1 (+) p2 ⇒ p3
    p1 (+) p3 ⇒ p2
    p2 (+) p3 ⇒ p1
```
[download]

which is all well and good. **But,** if we add the following symbols:

```
    j1  ::= john knows 'the morning star' equals 'the planet venus'
    j2  ::= john knows 'the evening star' equals 'the planet venus'
    j3  ::= john knows 'the morning star' equals 'the evening star'
```
[download]

we can **not** legally generate the following statements:

```
    j1 (+) p2 ⇒ j3
    p1 (+) j2 ⇒ j3
    j1 (+) p3 ⇒ j2
    p1 (+) j3 ⇒ j2
    j2 (+) p3 ⇒ j1
    p2 (+) j3 ⇒ j1
```
[download]

The word 'knows' makes the 'j' frame of reference referentially opaque. That means we can't assume that the 'p' statements are automatically true within the 'j' frame of reference.

So what does this have to do with functional programming? Two words: **dynamic scoping.** Granted the following is Perl code, but it obeys the constraint that every value remains constant within the scope of its variable:

```
    sub outer_1 {
        local ($x) = 1;
        return (inner ($_[0]));
    }

    sub outer_2 {
        local ($x) = 2;
        return (innner ($_[0]));
    }

    sub inner {
        my $y = shift;
        return ($x + $y);
    }
```

[download]

The function inner() is referentially opaque because it relies on a dynamically scoped variable. With a little work, we could replace $x with locally scoped functions and get the same result from code that passes the 'doesn't use assignment' rule with flying colors.

And that's just the trivial version. There's also the issue of quoting and defining equivalence between different representations of the same value. The following function defines equivalence between numbers and strings:

```
    sub equals {
        my ($str, $num) = @_;

        my %lut = (
            'one'   =>  1,
            'two'   =>  2,
            'three' =>  3,
            ...
        );

        if ($num == $lut{ $str }) {
            return 1;
        } else {
            return 0;
        }
    }
```

[download]

but even though equals('three',3) is true, length('three') does not equal length(3). Once again, we've killed referential transparency in a way that has nothing to do with assigning values to mutable storage.

The fact of the matter is, referential transparency isn't all that desirable a property for a programming language. It imposes such incredibly tight constraints on what you can do with your symbols that you end up being unable to do anything terribly interesting. And propagating the myth that functional programming allows universal substitutability is just plain evil, because it sets people up to have examples like these two blow up in their faces.

Functional programming defines the block as a frame of reference. All frames of reference are immediately visible from trivial inspection of the code. All symbols are substitutable within the same frame of reference. These things are GOOD! Learn them. Live them. Love them. Rejoice in them. Accept them for what they are and don't try to inflate them into something that sounds cooler but really isn't worth having.

Myth #4 - Functional programming doesn't support variable assignment

It's been said that C lets you shoot yourself in the foot.

It's said that C++ makes it harder, but when you do, you blow your whole leg off.

I'm personally of the opinion that functional programming makes it even harder to shoot yourself in the foot, but when you do, all that's left are a few strands of red goo dangling from the shattered remains of your brain pan.

Statements like the one above are the reason I hold that opinion.

Let's go back to Myth #2, shall we? According to Church's thesis, all programming languages are computationally equivalent. They all do the same things, and only the syntactic sugar is different.

In this case, the syntactic sugar is *so* different that people can end up using variable assignment without knowing they're doing it, all the while smugly assuming they're free from the evils of imperative programming.

To understand how that can happen, let's take a good hard look at what 'variable assignment' actually means. For the sake of discussion, I'm going to use the term 'lvalue' instead of 'variable', because an

lvalue is explicitly something to which you can assign values. Using that term, we can restate this myth as "functional programming doesn't support lvalues."

So.. what's an lvalue? In terms of implementation, it's a register where values are stored, but what is it functionally? How does the rest of the program see it?

Well, what the rest of the program sees is a symbol that gets associated with a series of different values.

So, theoretically, we could collect all the values that are stored in an lvalue during its life in the program, and store them in a list. Instead of saying just plain $x, we could look up the value $x[$t], where $t indicates the number of changes the value goes through before it reaches the value we want.

That fact defines the bridge between functional and imperative programming. We can simulate any lvalue with a list that contains the same sequence of values, and never violate the functional 'doesn't use assignment' rule.

But we can make our simulation even more interesting by realizing that each new value in the list is usually the result of a calculation applied to the previous value. The idea of building a new list by sequentially applying a list of operations to an initial value is a *very* functional programming idea. The following code uses that idea to implement an integer counter:

```
my $inc  = sub { return ($_[0] + 1) };
my $dec  = sub { return ($_[0] - 1) };
my $zero = sub { return (0) };

sub apply {
    my ($val, $func, @etc) = @_;

    if (@etc) {
        return ($val, apply ($func→($val), @etc));
    } else {
        return ($val, $func→($val));
    }
}

sub counter {
    return apply (0, @_);
}

my @ops = (
```

```
        $inc, $inc, $inc, $dec,
        $inc, $dec, $dec, $inc,
        $zero, $inc, $inc, $inc
    );

    print join (' ', counter (@ops));

    ----

    output == '0 1 2 3 2 3 2 1 2 0 1 2 3'
```

[download]

in about as functional a way as you can manage in Perl. A decent functional programmer would have no trouble porting that idea over to their functional language of choice.

Thing is, most people wouldn't think of code like that when they think '++', '--', '=0'.

Now, the list returned by counter() does have the bookkeeping-friendly feature that all the values are there simultaneously, in the same frame of reference. We could change that by dynamically scoping the list @ops, and distributing its contents across a whole series of evaluation contexts.

Even if all the values do exist in the same frame of reference, though, we still run into the same problems that make lvalues such a nuisance. If we tweak apply() so it only returns the final calculated value:

```
    sub apply {
        my ($val, $func, @etc) = @_;

        if (@etc) {
            return (apply ($func→($val), @etc));
        } else {
            return ($func→($val));
        }
    }
```

[download]

this code:

```
    my @x = ($inc, $inc, $inc);
    my @y = ($inc, $inc, $inc);

    if (3 == counter (@x)) {
        printf "%d equals %d.\n", counter (@x), counter (@y);
```

```
    }

    do {
        my @x = (@x, $inc);

        if (3 == counter (@y)) {
           printf "%d equals %d.\n", counter (@x), counter (@y);
        }
    }
```

[download]

demonstrates the same failure of substitutability that I used above, but in functional style. This version does have the advantage that the change from one frame of reference to another is immediately visible, but you can still screw around with the values in ways that aren't immediately obvious. This is the simple form of the problem. As with the lvalue version above, you can obfuscate it until your head explodes.

And that brings me to the way functional languages politely abuse the principle of lazy evaluation to handle user input. They pull basically the same kind of trick by treating the user input stream as a list where all the values are theoretically defined as soon as the program starts, but we don't have to prove it for any specific item of input until the user actually types it in. We can define the @ops list in terms of the (theoretically) defined user input list, and produce what amounts to an integer counter that responds to user input.

I don't object to the fact that functional programming allows these things to happen. I JUST WANT PEOPLE TO BE AWARE OF IT, AND STOP TREATING FP LIKE SOME KIND OF MAGIC BULLET BECAUSE IT SUPERFICIALLY APPEARS NOT TO DO THINGS THAT IT ACTUALLY DOES QUITE HAPPILY, THANK YOU VERY MUCH.

Winding down

Okay.. I feel better, now.

So that's my current spin on functional programming. 'Blocks equal frames of reference'? Way cool. Big thumbs up. 'Capacity to hide value storage in places where even the gurus don't go if they don't have to'? Not so cool. Beware of arbitrarily multicontextual dog. As a worldview and way of thinking about programming? Love it. As a body of advocacy? Please, please, puh-leeeease learn enough about it to explain it to the newbies without lying to them in ways that will make their heads go boom some time in the future.

Make their heads go boom now. It's more fun to watch, and it keeps them out of my hair.

Considered by bradcathey - Tone down language, like title: FP, pros and cons
Unconsidered by castaway - Keep/Edit/Delete: 34/11/0

Back to Meditations