Raganwald 2008

(*This is a snapshot of my old weblog. New posts and selected republished essays can be found at* raganwald.com. *A reformatted and cleaned up version of this post can be found* here.)

Sunday, March 11, 2007

Why Why Functional Programming Matters Matters

I recently re-read the amazing paper Why Functional Programming Matters (“WhyFP”). Although I thought that I understood WhyFP when I first read it a few years ago, when I had another look last weekend I suddenly understood that I had missed an important message.[1]

Now obviously (can you guess from the title?) the paper is about the importance of one particular style of programming, functional programming. And when I first read the paper, I took it at face value: I thought, “Here are some reasons why functional programming languages matter.”

On re-reading it, I see that the paper contains insights that apply to programming in general. I don’t know why this surprises me. The fact is, programming language design revolves around program design. A language’s design reflects the opinions of its creators about the proper design of programs.

In a very real sense, the design of a programming language is a strong expression of the opinions of the designer about good programs. When I first read WhyFP, I thought the author was expressing an opinion about the design of good programming languages. Whereas on the second reading, I realized he was expressing an opinion about the design of good programs.

Can we add though subtraction?

> It is a logical impossibility to make a language more powerful by omitting features, no matter how bad they may be.

Is this obvious? So how do we explain that one reason Java is considered “better than C++” is because it omits manual memory management? And one reason many people consider Java “better than Ruby” is because you cannot open base classes like `String` in Java? So no, it is not obvious. Why not?

The key is the word *better*. It’s not the same as the phrase *more powerful*.[2] The removal or deliberate omission of these features is an expression about the idea that programs which do not use these features are better than programs which do. Any feature (or removal of a feature) which makes the programs written in the language better makes the language better. Thus, it *is* possible to make a language “better” by removing features that are considered harmful,[3] if by doing so it makes programs in the language better programs.

In the opinion of the designers of Java, programs that do not use `malloc` and `free` are safer than those that do. And the opinion of the designers of Java is that programs that do not modify base classes like `String` are safer than those that do. The Java language design emphasizes a certain kind of safety, and to a Java language designer, safer programs are better programs.

“More powerful” is a design goal just like “safer.” But yet, what does it mean? We understand what a safer language is. It’s a language where programs written in the language are safer. But what is a “more powerful” language? That programs written in the language are more powerful? What does that mean? Fewer symbols (the “golf” metric)?

WhyFP asserts that you cannot make a language more powerful through the removal of features. To paraphrase an argument from the paper, *if removing harmful features was useful by itself, C and C++ programmers would simply have stopped using* `malloc` *and* `free` *twenty years ago*. Improving on C/C++ was not just a matter of removing `malloc` and `free`, it was also a matter of adding automatic garbage collection.

> This space, wherein the essay ought to argue that Java compensates for its closed base classes by providing a more powerful substitute feature, left intentionally blank.

At the same time, there is room for arguing that some languages are improved by the removal of harmful features. To understand why they may be improved but not more powerful, we need a more objective definition of what it means for a language to be “more powerful.” Specifically, what quality does a more powerful programming language permit or encourage in programs?

When we understand what makes a program “better” in the mind of a language designer, we can understand the choices behind the language.

Factoring

Factoring a program is the act of dividing it into units that are composed to produce the working software.[4] Factoring happens as part of the design. (*Re*-factoring is the act of rearranging an existing program to be factored in a different way). If you want to compare this to factoring in number theory, a well designed program has lots of factors, like the number $3,628,800$ ($10!$). A Big Ball of Mud is like the number $3,628,811$, a prime.

Composition is the construction of programs from smaller programs. So factoring is to composition as division is to multiplication.

Factoring programs isn’t really like factoring simple divisors. The most important reason is that programs can be factored in orthogonal ways. When you break a program into subprograms (using methods, subroutines, functions, what-have-you), that’s one axis of factoring. When you break an a modular program up into modules, that’s another, orthogonal axis of factoring.

Programs that are well-factored are more desirable than programs that are poorly factored.

> In computer science, **separation of concerns** (SoC) is the process of breaking a program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program.
>
> SoC is a long standing idea that simply means a large problem is easier to manage if it can be broken down into pieces; particularly so if the solutions to the sub-problems can be combined to form a solution to the large problem.
>
> The term separation of concerns was probably coined by Edsger W. Dijkstra in his paper On the role of scientific thought.

—Excerpts from the Wikipedia entry on Separation of Concerns

Programs that separate their concerns are well-factored. There’s a principle of software development, responsibility-driven design. Each component should have one clear responsibility, and it should have everything it needs to carry out its responsibility.

This is the separation of concerns again. Each component of a program having one clearly defined responsibility means each concern is addressed in one clearly defined place.

> Let’s ask a question about Monopoly (and Enterprise software). Where do the rules live? In a noun-oriented design, the rules are smooshed and smeared across the design, because every single object is responsible for knowing everything about everything that it can ‘do’. All the verbs are glued to the nouns as methods.

—My favourite interview question

In a game design where you have important information about a rule smeared all over the object hierarchy, you have very poor separation of concerns. It looks at first like there’s a clear factoring “Baltic Avenue has a method called `isUpgradableToHotel`,” but when you look more closely you realize that every object representing a property is burdened with knowing almost all of the rules of the game.

The Seasoned Schemer is devoted to the myriad uses of first class functions. This book is approachable and a delight to read, but the ideas are provocative and when you close the back cover you will be able to compose programs from functions in powerful new ways.

The concerns are not clearly separated: there’ s no one place to look and understand the behaviour of the game.

Programs that separate their concerns are better programs than those that do not. And languages that facilitate this kind of program design are better than those that hamper it.

Power through features that separate concerns

One thing that makes a programming language “more powerful” in my opinion is the provision of more ways to factor programs. Or if you prefer, *more axes of composition*. The more different ways you can compose programs out of subprograms, the more powerful a language is.

Do you remember Structured Programming? The gist is, you remove `goto` and you replace it with well-defined control flow mechanisms: some form of subroutine call and return, some form of selection mechanism like Algol-descendant `if`, and some form of repetition like Common Lisp’ s `loop` macro.

Dijkstra’ s view on structured programming was that it promoted the separation of concerns. The factoring of programs into blocks with well-defined control flow made it easy to understand blocks and rearrange programs in different ways. Programs with indiscriminate jumps did not factor well (if at all): they were difficult to understand and often could not be rearranged at all.

Structured 68k ASM programming is straightforward in theory. You just need a lot of boilerplate, design patterns, and the discipline to stick to your convictions. But of course, lots of 68k ASM programming in practice is only partially structured. Statistically speaking, 68k ASM is not a structured programming language even though structured programming is possible in 68k ASM.

Structured Pascal programming is straightforward both in theory and in practice. Pascal facilitates separation of concerns through structured programming. So we say that Pascal “is more powerful than 68k ASM” to mean that in practice, programs written in Pascal are more structured than programs written in 68k ASM because Pascal provides facilities for separating concerns that are missing in 68k ASM.

For example: working with lists

Consider this snippet of iterative code:

```
int numberOfOldTimers = 0;
for (Employee emp: employeeList) {
    for (Department dept: departmentsInCompany) {
        if (emp.getDepartmentId() == dept.getId() && emp.getYearsOfService() > dept.getAge()) {
            ++numberOfOldTimers;
        }
    }
}
```

This is an improvement on older practices.[5, 6] For one thing, the `for` loops hide the implementation details of iterating over `employeeList` and `departmentsInCompany`. Is this better because you have less to type? Yes. Is it better because you eliminate the fence-post errors associated with loop variables? Of course.

But most interestingly, you have the beginnings of a *separation of concerns*: how to iterate over a single list is separate from what you do in the iteration.

> Try calling a colleague on the telephone and explaining what we want as succinctly as possible. Do you say "We want a loop inside a loop and inside of that an if, and…"? Or do you say "We want to count the number of employees that have been with the company longer than their departments have existed."

One problem with the `for` loop is that it can only handle one loop at a time. We have to nest loops to work with two lists at once. This is patently wrong: there's nothing inherently nested about what we're trying to do. We can demonstrate this easily: try calling a colleague on the telephone and explaining what we want as succinctly as possible. Do you say "We want a loop inside a loop and inside of that an if, and…"?

No, we say, "We want to count the number of employees that have been with the company longer than their departments have existed." There's no discussion of nesting.

In this case, a limitation of our tool has caused our concerns to intermingle again. The concern of "How to find the employees that have been with the company longer than their departments have existed" is intertwined with the concern of "count them." Let's try a different notation that separates the details of *how to find* from the detail of *counting what we've found*:

```
old_timers = (employees * departments).select do |emp, dept|
  emp.department_id == dept.id && emp.years_of_service > dept.age
end
```

```
number_of_old_timers = old_timers.size
```

Now we have separated the concern of finding from counting. And we have hidden the nesting by using the * operator to create a Cartesian product of the two lists. Now let’ s look at what we used to filter the combined list, `select`. The difference is more than just semantics, or counting characters, or the alleged pleasure of fooling around with closures.

I’ m not a Haskell user (yet), but The Haskell School of Expression: Learning Functional Programming through Multimedia has received rave reviews and comes with solid recommendations. It’ s on my wish list if you’ re feeling generous!

* and `select` facilitates separating the concerns of how to filter things (like iterate over them applying a test) from the concern of what we want to filter. So languages that make this easy are more powerful than languages that do not. In the sense that they facilitate additional axes of factoring.

The Telephone Test

Let’ s look back a few paragraphs. We have an example of the “Telephone Test:” when code very closely resembles how you would explain your solution over the telephone, we often say it is “very high level.” The usual case is that such code expresses a lot more *what* and a lot less *how*. The concern of what has been very clearly separated from the concern of how: you can’ t even *see* the how if you don’ t go looking for it.

In general, we think this is a good thing. But it isn’ t free: somewhere else there is a mass of code that supports your brevity. When that extra mass of code is built into the programming language, or is baked into the standard libraries, it is nearly free and obviously a Very Good Thing. A language that doesn’ t just separate the concern of how but does the work for you is very close to “something for nothing” in programming.

But sometimes you have to write the *how* as well as the *what*. It isn’ t always handed to you. In that case, it is still valuable, because the resulting program still separates concerns. It still factors into separate components. The components can be changed.

I recently separated the concern of describing “how to generate sample curves for some data mining” from the concern of “managing memory when generating the curves.” I did so by writing my own lazy evaluation code (Both the story and the code are on line). Here’ s the key “what” code that generates an infinite list of parameters for sample beziér curves:

```
def magnitudes
  LazyList.binary_search(0.0, 1.0)
end

def control_points
```

```
  LazyList.cartesian_product(magnitudes, magnitudes) do |x, y|
    Dictionary.new( :x => x, :y => y )
  end
end

def order_one_flows args = {}
  height, width = (args[:height] || 100.0), (args[:width] || 100.0)
  LazyList.cartesian_product(
      magnitudes, control_points, control_points, magnitudes
  ) do |initial_y, p1, p2, final_y|
    FlowParams.new(
      height, width, initial_y * height,
      CubicBezierParams.new(
        :x => width,          :y => final_y * height,
        :x1 => p1.x * width,  :y1 => p1.y * height,
        :x2 => p2.x * width,  :y2 => p2.y * height
      )
    )
  end
end
```

That’ s it. Just as I might tell you on the phone: “Magnitudes” is a list of numbers between zero and one created by repeatedly dividing the intervals in half, like a binary search. “Control Points” is a list of the Cartesian product of magnitudes with itself, with one magnitude assigned to `x` and the other to `y`. And so forth.

I will not say that the sum of this code and the code that actually implements infinite lists is shorter than imperative code that would intermingle loops and control structures, entangling *what* with *how*. I will say that it separates the concerns of what and how, and it separates them in a different way than `select` separated the concerns of what and how.

So why does “Why Functional Programming Matters” matter again?

The great insight is that better programs separate concerns. They are factored more purely, and the factors are naturally along the lines of responsibility (rather than in Jenga piles of `abstract virtual base mixin module class proto_ extends private implements`). Languages that facilitate better separation of concerns are more powerful in practice than those that don’ t.

WhyFP illustrates this point beautifully with the same examples I just gave: first-class functions and lazy evaluation, both prominent features of modern functional languages like Haskell.

WhyFP’ s value is that it expresses an opinion about what makes programs better. It backs this opinion up with reasons why modern functional programming languages are more powerful than imperative programming languages. But even if you don’ t plan to try functional programming tomorrow, the lessons about better programs are valuable for your work in *any* language today.

That’ s why Why Functional Programming Matters matters.

1. And now I' m worried: what am I *still* missing?

2. Please let' s not have a discussion about Turing Equivalence. Computer Science "Theory" tells us "there' s no such thing as more powerful." Perhaps we share the belief that *In theory, there' s no difference between theory and practice. But in practice, there is.*

3. I am not making the claim that *I* consider memory management or unsealed base classes harmful, but I argue that there exists at least one person who does.

4. The word "factor" has been a little out of vogue in recent times. But thanks to an excellent post on reddit, it could make a comeback.

5. So much so that we won' t even bother to show what loops looked like in the days of `for (int i = 0; i < employeeList.size(); ++i)`.

6. Another organization might merge employees and departments, or have each department "own" a collection of employees. This makes our example easier, but now the *data* doesn' t factor well. Everything we' ve learned from databases in the last forty years tells us that we often need to find new ways to compose our data. The relational model factors well. The network model factors poorly.

Labels: java, lispy, popular, ruby

¶ 10:21 PM

Comments on "*Why Why Functional Programming Matters Matters*" :

Regarding the "power" footnote:
Why not invent a term such as "factorability", since that's what you spend most of the post talking about? Power is power, and Church and Turing have pretty much dealt with that. Let's not overload the word power with other concepts that make it too hard to tell what we're talking about. At the very least, pick something a bit closer. "Expressiveness," perhaps.
# posted by Scott : 1:53 AM

Hi Reginald,

Just a Ruby comment. In your code sample, the operator * is used for 2 arrays as a cartesian product. While this is trivial to add it to the Array class, I don't find it the standard ruby distribution 1.8.5. Did you implement it or did you use an extension lib for the Array class?
# posted by Eric : 1:57 AM

In the Ruby version of counting old timers, am I misreading something or shouldn't the first "employees_and_departments" really be an "old_timers"?
# posted by Vineet : 2:50 AM

Consider the APL and J languages, and the latest from that camp, Arthur Whitne's K (at http://kx.com/). It's a programming language that dominates the golf course, and has notoriety as less readable than perl because of that; But most importantly, it eschews standard notions of OO and many FP languages. Instead, it focuses on 40 or so basic operations, and some 20 ways to compose them. The result is really amazing -- a functional (thus, safe) language, very easy to use once you get the hang of it, with an incredibly quick interpreter that often outdoes comparable C code, in a 120K executable.
# posted by Anonymous : 3:54 AM

Scott: CHurch and Turning have dealt with *computability* in *theory*.

This essay discusses power in practice. See the footnote above.
# posted by Reginald Braithwaite : 7:34 AM

Eric:

The code for * is in the lazy lists class mentioned above (the source is here).

There is an implementation for arrays as part of the Ruby Facets class, it is the combinations method added to Enumerable.

My version happens to work with infinite lists. And by strange coïincidence, the code above does not actually declare that the given lists are Arrays...

Perhaps they are also lazy?
# posted by Reginald Braithwaite : 8:03 AM

Hi Reg,

Another thoughtful post, thank you. Being an admirer, I also appreciate your reference to Edsger W. Dijkstra's Separation of Concerns.

In reading a number of your posts, I see a common theme of talking about the problem of the semantic distance between the language of the problem domain and the language used to implement a solution, without explicitly saying that you are talking about this distance. Your example of contrasting two implementations of counting the employees is a wonderful demonstration of this distance. My question is why do you not come out and specifically say that this is what you are talking about or to turn around your question: what am I missing?
# posted by Norbert : 8:53 AM

To abridge the discussion that has been going on at Reddit (http://programming.reddit.com/info/19leo/comments/c19lwv) Reg's argument seems to involve a measure of power for a language that has to do with transformations that are possible to do once an idea is expressed in that language.

My position is that I would prefer a measure of power that increased depending on how few transformations are needed to move the proposed solution from an idea in your head to an expression in the language.

It'll take me probably a week to get this down coherently no my web site, but once I do so it will be at http://www.kirit.com/Measuring%20the%20power%20of%20programming%20languages
# posted by Kirit : 9:16 AM

-
I think the established word that you are looking for is 'compositionality' or 'decompositionality'.

There are many advantage to compositionality, many of which you list. But there is one major disadvantage as well: efficiency. Compositional optimisation, i.e. optimising the various parts of a program then plugging them together, will always be strictly less efficient that global optimisation, which always results in spaghetti code.
# posted by Dan : 10:05 AM

-
Another great article.

Just a quick note re: your nested looping antidote example. Ruby supports some light destructuring out of the box, so you could have written

```
old_timers = (employees * departments).select do |emp,dept|
  emp.department_id == dept.id && emp.years_of_service > dept.age
end
number_of_old_timers = old_timers.size
```

instead. Pretty trivial, but makes the code even nicer to read.
# posted by Justin : 12:54 PM

-
Ruby supports some light destructuring out of the box

Hey, I didn't know you could do that without splattage. Thanks!
# posted by Reginald Braithwaite : 12:58 PM

-
Reginald yet another gem from you.

"So how do we explain that one reason Java is considered “better than C++” is because it omits manual memory management?"

As a side note, a wrong perception is always given that Java removes/omit memory management. In fact it "replaces" it with a Garbage Collector.

Which does not mean you are not responsible for memory management any more. We are sill responsible for memory management but a different way. I blogged about it some time back
Memory management is STILL your responsibility

Off course you mention the Java example to explain a lot different thing.

Cool post keep them coming!
# posted by TH : 2:57 PM

TH:

Which does not mean you are not responsible for memory management any more. We are sill responsible for memory management but a different way.

Funny thing, but when we started selling JProbe Memory Debugger, we had to pound that message into the market over and over and over.

One of our favourite "talking points" was a list of four kinds of memory leaks in Java programs.

The thought at the time was that "Java programs do not leak," so that was always a fun topic.
# posted by Reginald Braithwaite : 3:04 PM

Regarding the code samples wrt iterating over the lists:

The code you show is not object oriented at all, it's relational and hence your problems. If you designed your DAO and logic objects properly you'd just write something like:

```
int oldTimers = 0;
for (Department dept: departments) {
for (Employee emp: dept.getEmployees()) {
if (emp.getYearsOfService() > dept.getAge()) {
++oldTimers;
}
}
}
```

In fact, your Ruby version makes things possibly even worse. Unless there is some lazy magic there, your Cartesian product will needlessly eat lots of memory and get garbage collected just afterwards (of course, your Java version might needlessly connect to a database behind, so it's hard to decide which is really worse). In the Ruby version you're also creating an unneeded list of oldtimers, but its only fate is to yield its length and get eaten as well.

But whatever happens, both your Java and Ruby versions needlessly iterate over dept_count*emp_count pairs, while properly designed OO version iterates only emp_count times over exactly those pairs (dept, emp) which make sense.

I really can't believe that functional design cannot do that.
# posted by albert bulawa : 5:49 PM

Albert:

If you designed your DAO and logic objects properly

In other words, if you **pushed joining two lists somewhere else**?

So what is the real argument? Could it be that *a properly designed system separates the mechanism of joining (by putting it in a DAO that talks to an RDBMS) from what we want to do with the result*?

Are we arguing **the exact same point**, only using different examples?

I think this is a case where perhaps you would have preferred to see some production code to a simple example?
# posted by Reginald Braithwaite : 5:58 PM

For monopoly, I'd imagine that it'd be better to make a new static class called Referee, who'd know all the rules. Then you would consult with the Referee for any moves you made. Also, any tweaks to the rules (such as whether or not landing on Free Parking paid out of a pot) could be kept in one place.
# posted by Sean : 6:04 PM

As promised, Kirit has followed up:

Measuring the power of programming languages.

I like it, thanks!
# posted by Reginald Braithwaite : 10:48 PM

*But whatever happens, both your Java and Ruby versions needlessly iterate over dept_count*emp_count pairs, while properly designed OO version iterates only emp_count times over exactly those pairs (dept, emp) which make sense.*

Indeed. In Ruby a good solution would probably end up looking something like the one you've given. I'd say that is a better separation of concerns as well, as either an employee should know their department or a department should know their employees.

That said, a similarly written solution in Haskell (from my understanding) need not hold the list of all employees and departments.

This would be a rather natural way of writing that program, using a list comprehension.

```
length [Nothing | emp <- employees,
dept <- departments,
getDeptId emp == getId dept,
getYearsOfService emp > getDeptAge dept]
```
# posted by Peter : 11:46 PM

Oy, debating the merits of object designs is interesting, but really should be over at the Monopoly post... which is about design.

But we're programmers, and if we see an essay with 500 words and five lines of code, we'll give the words a few minutes and then we'll spend hours talking about the five lines of code.

Remember, the code is to illustrate something about factoring.

Okay, so:

First, it's easy to have an employee hold a reference to a department. If I did that, I could change the example to count departments that have old timers.

Whats the difference? Now you have a problem of going from department to employee.

Should departments hold a collection of employees? I don't think so. I'm okay with a *convenience method* that returns such a collection, but behind the scenes, something else has to manage that collection.

Although we say "department has many employees" aloud, if we actually make a department a container for employees, we have two different entities sharing a responsibility.

What happens when we write "emp.department = new_dept"? You have to update two different objects, the employee and the department. Likewise, if you write "dept.employees << new_hire_employee", you have to update two objects again.

You end up with employees responsible for managing departments AND departments responsible for managing employees. This is a code smell: there should be exactly one place where the relationship between the two is updated.

Of course, you can write those exact statements in Rails. Is that bad? No, because Rails virtualizes those methods.

In actual fact, there is one thing in Rails that manages the relationship, a table in a database. The methods are provided for convenience. And if you change the relationship, the methods change for you automatically: you don't have to update the methods in two places.

So while Rails looks like it give you employees that have a reference to a department and departments that have a collection of employees, the actual mechanism for managing the relationship is factored out of employees and out of departments.

Which, I think, i entirely in keeping with the actual point of the words of this post, about factoring how away from what.
# posted by Reginald Braithwaite : 7:49 AM

Reginald: Weren't we talking about SoC?! That's exactly what you missed from that example. This code should not know that there is any deptid and that this is what links an employee to their department. Depending on what application should really do, this Department.getEmployees() method could return a pre-created list, join lists in background, do a select to a single RDBMS column, whatever. But that does not belong in this code snippet, and neither do fields which may be required by backend storage but are meaningless at this business logic level.

I may miss your point, so a real-life code sample could probably help.
# posted by albert bulawa : 10:08 AM

"a select to a single RDBMS column"

Of course, I meant a table.
# posted by albert bulawa : 10:10 AM

Albert:

Your suggestion that the methods be proxies for some other thingie is exactly what I would do in "real life."

I would probably use some metaprogramming (or annotation or whatever the language in question calls code that writes code).

But you know, my example is still entirely valid and concise. There is a better separation of concerns using Cartesian product and select than there is iterating when you wish to perform a join and filter.

I think you are bothered by the example of employees and departments because you probably imagine that the relationship is something permanent and ought to be reflected directly in the Entity-Relationship Model.

If you like, you can consider a case with a much more transient relationship. For example:

We have a list of coffee shops, graded by quality and geocoded for location. We have a list of potential offices where we can locate a new start up.

We are looking for locations that have at least two good coffee shops within walking distance from them.

I am tempted to ask whether we can agree that one style of code separates the concern better than the other.

In fact, that is not the question. The **real** question is, can we agree that some styles of code separate concerns better than others?

As long as we agree that some code separates concerns better than others, that's all that matters.

You can really pick any two chunks of code that do the same thing and point out which one does a better job. The thesis is that some code separates concerns better than others, and that some tools facilitate that separation better than others.
# posted by Reginald Braithwaite : 10:34 AM

In response to your disclaimer about Turing equivalence, and your comment to Scott: you *can* make **formal** claims about expressive power beyond Turing equivalence. There's a classic paper by Felleisen (co-author of the *Seasoned Schemer*, which you cite) about this. Read, enjoy, and be enlightened.
# posted by Shriram Krishnamurthi : 10:19 PM

<< Home
Reg Braithwaite

Recent Writing
Homoiconic Technical Writing / raganwald.posterous.com

Books
What I ‘ve Learned From Failure / Kestrels, Quirky Birds, and Hopeless Egocentricity

Share
rewrite_rails / andand / unfold.rb / string_to_proc.rb / dsl_and_let.rb / comprehension.rb / lazy_lists.rb

Beauty
IS-STRICTLY-EQUIVALENT-TO-A / Spaghetti-Western Coding / Golf is a good program spoiled / Programming conventions as signals / Not all functions should be object methods

The Not So Big Software Design / Writing programs for people to read / Why Why Functional Programming Matters Matters / But Y would I want to do a thing like this?

Work
The single most important thing you must do to improve your programming career / The Naïve Approach to Hiring People / No Disrespect / Take control of your interview / Three

tips for getting a job through a recruiter / My favourite interview question

Management
Exception Handling in Software Development / What if powerful languages and idioms only work for small teams? / Bricks / Which theory fits the evidence? / Still failing, still learning / What I’ ve learned from failure

Notation
The unary ampersand in Ruby / (1..100).inject(&:+) / The challenge of teaching yourself a programming language / The significance of the meta-circular interpreter / Block-Structured Javascript / Haskell, Ruby and Infinity / Closures and Higher-Order Functions

Opinion
Why Apple is more expensive than Amazon / Why we are the biggest obstacles to our own growth / Is software the documentation of business process mistakes? / We have lost control of the apparatus / What I’ ve Learned From Sales I, II, III

Whimsey
The Narcissism of Small Code Differences / Billy Martin’ s Technique for Managing his Manager / Three stories about The Tao / Programming Language Stories / Why You Need a Degree to Work For BigCo

History
06/04 / 07/04 / 08/04 / 09/04 / 10/04 / 11/04 / 12/04 / 01/05 / 02/05 / 03/05 / 04/05 / 06/05 / 07/05 / 08/05 / 09/05 / 10/05 / 11/05 / 01/06 / 02/06 / 03/06 / 04/06 / 05/06 / 06/06 / 07/06 / 08/06 / 09/06 / 10/06 / 11/06 / 12/06 / 01/07 / 02/07 / 03/07 / 04/07 / 05/07 / 06/07 / 07/07 / 08/07 / 09/07 / 10/07 / 11/07 / 12/07 / 01/08 / 02/08 / 03/08 / 04/08 / 05/08 / 06/08 / 07/08 /