

sealed abstract

I write software 

- [home](#)
- [business](#)
- [Code](#)
- [iphone](#)
- [rants](#)
- [Things I'm working on](#)

[RSS Feed](#) 

Broken Promises

01 April 2013 by [Drew Crawford](#) Published in: [Code](#), [rants](#) [7 comments](#) 

James Coglan [published an article the other day](#) about how node.js missed the boat with promises. I don't know much about node.js, but I do know about promises. And they didn't miss much of a boat.

So I'm an iOS developer, and for reasons outside the scope of this blog post, you have a lot of concurrency problems these days in iOS. And I said to myself, "You know what we need? Promises. Those will solve all our problems." Well, they didn't. I spent way too much time trying to make them solve the problems, and it was a lot more of a half-baked idea than it seemed like going in.

Now I should say here, and you'll see at the end, promises themselves are *fine*. The trouble is with the *evangelists*. The war between promises and callbacks is kind of like the war between while loops and for loops: sometimes you want one and other times you want the other, but they're **not really that much different**. The problem is that this is a little hard to see at the beginning, and you have the promise evangelists saying things like:

I hope to dismiss the misunderstanding that promises are about having cleaner syntax for callback-based async work. They are about modeling your problem in a fundamentally different way; they go deeper than syntax and actually change the way you solve problems at a semantic level.

Just as an introductory jab, this is **false by definition** because **that is how promises are defined**. Just ask [Baker and Hewitt](#), two of the earliest scientists on the scene:

...by creating for each process a *future*, which is a promise to deliver the value of that subexpression at some later time, if it has a value.

Just like a callback, no? Or, for example, you can ask the original inventors of promises and futures, [Friedman & Wise](#):

It is our thesis that the fields of a newly allocated record can be filled with a structure representing the suspended evaluation of that argument... If all other elementary functions are able to detect these suspensions and to force evaluation only at the time that the value is genuinely critical to the course of the computation (necessary to the value of the main function), then **the results are the same as those of a strict evaluation scheme** whenever both converge.

But what the promise evangelists *really mean* when they say "promises are about modeling your problem in a fundamentally different way" is they mean promises **plus some hypothetical library** can do this. They mean that promises are a *convenient way to specify*, say, to a "control flow engine library", how your code should execute. But the trouble with hypothetical libraries is that they are hypothetical. So let's go on an imaginary journey writing that library. Except it's less of a hypothetical library, and more *the actual library that I tried, and failed, to write*. I have been to the mountaintop, and it wasn't much to write home about really.

I don't want to rain too hard on James (if you're reading this, James, lunch is on me the next time you're in Austin), because he seems like a nice guy, but his blog post is probably the most cogent blog post on the subject of promises and the one that I would have written in Objective-C syntax two months ago.

Except now I'm writing this one, saying quite nearly the opposite thing. So think of it like me trying to convince *two-months-ago-me* why *I* am crazy, because that's really what I'm doing.

The memory problem

Getting the result out of a callback- or event-based function basically means "being in the right place at the right time". If you bind your event listener after the result event has been

fired, or you don't have code in the right place in a callback, then tough luck, you missed the result.

If callbacks mean being in the right place at the right time, then promises mean "having enough memory to keep data around on the off chance that you need it more than once." I don't know about what kind of environments in which people use node.js, but in the iOS world there's nowhere *near* enough memory to hang onto things on the off chance that we might need them later on. Just to put some numbers on it, you can fit maybe 3 camera-quality photos in memory at the same time. So if you have promises for four photos, your app quits unexpectedly. If you have a for loop that executes for each photo, and you build a promise for each one, you're **done**.

Solving the "what if we run out of memory problem" effectively reduces to building a large cache that purports to drop some data on the floor if maybe we don't need it. But this is a **hard** problem in part because some operations have side effects (like deletes or updates) and if you drop that data you can never get it back, and in part because the caching problem is one of the oldest problems in all of computer science.

There's another simple case that's even hairier. Suppose you have a rather large file, and you want to get a string of "[file] + [single character]". So you invent a promise to get the file, and you invent another promise to append the character. Now, what do we keep in memory? The file, the character, **and** the concatenation? That's crazy. And so you invent ridiculous semantics like "in-place mutation promises". But what if somebody needed the string of the original file? Well okay, let's store the two inputs, and recompute the *concatenation* if we need it. But what if we're doing an arbitrary slow computation instead of concatenation? And so on. I mean basically you need a garbage collector (to understand what promises are still in use and who uses them), **as well as** a time and memory profiler (to figure out what is cheap to dump and what isn't) **as well as** special semantics for side effect operations that we can never run again. It's madness.

But don't take my word for it. What is probably the [second-or-so paper](#) ever written on promises (in the first handful anyway), by the originators Friedman & Wise, says:

The system we have described... depends only on **reference counts** for memory management of the space used to store both environment and user structure.

Reference counts being, in this case, a pseudo-garbage-collection technique. But reference counting alone was rapidly found to be unsatisfactory for many environments. Baker & Hewitt write, as the third group of authors to speak on the subject:

We discuss an approach to a problem that arises in this context: futures which were thought to be relevant when they were created become irrelevant through not being needed later in the computation... The solution we propose is that of **incremental garbage collection**.

So there you go. Here we are, in the infancy of promises, where just a handful of people in the world have ever heard about them, and there are maybe three implementations of them that exist. And **already** we have **two** proposals about the memory problem. That's how fundamental it is.

And it turns out the problem has no good solution. Well it has one good solution: garbage collection. And this works great if you are tinkering around with the LISP interpreter (which is what the researchers were doing). But V8's garbage collector is entirely opaque to ordinary JS programs, and at any rate is designed with a "[run eventually](#)" philosophy that is entirely the opposite of what you want for futures: **run often**. So at minimum, you would need to design a special JS engine that has a special aggressive GC for promises in order to get any kind of reasonable performance. As I understand it, this is not a priority for the future of JavaScript.

Meanwhile, in iOS world, we don't have garbage collection. And for various different reasons, that are far outside the scope of this blog post, it's considered [infeasible as a practical matter](#) by the field experts to ship a good general-purpose garbage collector for iOS, despite the fact that many are trying very hard. And if "step 1" in the Great Plan for Promises is to solve a problem that is for the most part considered unsolved on the platform, you're gonna have a bad time.

Which leads us to:

The composition problem

The evangelists say things like:

a list containing promises is itself a promise that resolves when all its elements are resolved.

And a snake pit of problems opens before you so fast that you can't even enumerate all the ways this is going to suck:

1. How many items are in the list? I'll give you a hint: the answer involves waiting for the list to show up in our function. You might not think this is so terrible, but a **great number of perfectly ordinary things** depend on knowing how many elements there are. For example, **drawing the scrollbar in the user interface** requires knowing how many items are in the list, and what is the length of each item (e.g. if they are variably-sized). This forces you either to load the complete list of items (which is slow for long lists) or to design really strange list implementations that load the underlying elements in batches and try and tune the batch size.
2. What if the computation that produced list elements #1 and #3 are side effect-ful and can't be re-run, but items #2 and #4 are large and we're running out of memory? Do we purge the list or no?

- Or do we purge only part of it?
3. Are lists that hold the same promises the same list instance, or different list instances?
 4. If they are different list instances, how do we know not to evaluate the underlying promises over and over for each list?
 5. If they are the same list instance, how do we remember to recycle an existing list instance when somebody constructs a new one with exactly the same elements?
 6. How can we solve problems 4-6 in a way that doesn't allocate irreclaimable memory every time somebody makes a list?
 7. And if we let you turn, for example, a lambda into a promise, how do we define equality on lambdas? Are lambdas that run the same code the same instance, or different instances? And when we put "the same" or "two different" lambdas in a list, what happens?

I didn't ask any of these questions when I was super excited about promises. And I was not content with just "lists are promises for their elements", I **ran** with the idea. Why stop at lists! Promises in a box! Promises with a fox! Attributes are promises! Arrays are promises! Everything is a promise! And you would get code like this:

```
1 | bump = foo.bar["baz"][12].bap("123, 456")
```

Where bump, foo, bar, the array access, and the method call were all promises. And then at some ridiculously later time:

```
1 | actually_print_to_terminal(bump); //Exception: 12 is out of bounds  
for an empty array
```

What? What array? Well, the one in some other function that you didn't write. You see, some function somebody wrote thought it was getting an array of 16 elements, but element #3 failed to load from the network. ("Network? what network?") Oh yeah, "bar" is actually composed of a bunch of promises somewhere else... one of them goes out to network, maybe. The amount of exceptions that *totally unrelated code can raise is literally unbounded*.

But it's not just the **errors** that are unbounded. It's also the **code that runs** when you do "trivial" things like ask how many elements are in a list. For example, you will find yourself (and this was many, many hours of my life) wondering "Why does logging element #12 take 36 seconds and the others are instantaneous?" The answer is because something you needed fell out of cache and we have to go to network to get it. What was it? That's a very good question. The code that does it is a long, long way from the place that you are looking. Good luck, adventurer.

The parallelization problem

The evangelists say things like:

Using graphs of dependent promises lets you skip step B altogether. You write code that expresses the task dependencies and let the computer deal with control flow.

This is another one of those ideas that you could have copied and pasted from the blog post I was writing two months ago. Unfortunately it's **very very wrong**.

The trouble starts with shared resources. If you have a promise that does:

```
1 | db = open_database("db");  
2 | all_the_objects_count = db.count();  
3 | close(db);  
4 | return all_the_objects_count;
```

And maybe someone else is **inserting** an object while this is happening, then our count might be invalid, or the database might be in an absurd state and an exception is raised. Well the fix is trivial:

```
1 | lock("using_the_database");  
2 | // snip  
3 | unlock("using_the_database");
```

Easy, right! Well, it's easy until you have two locks:

```
1 | lock("using_the_database");  
2 | lock("using_the_ui");  
3 | lock("using_the_network");  
4 | //snip ...  
5 | unlock("using_the_network");  
6 | unlock("using_the_ui");  
7 | unlock("using_the_database");
```

Because now, something potentially as simple as printing an object, **can deadlock your program**. And you have **no idea why**.

But this is easy enough to solve, right? You just declare up front what locks you want to acquire, and let the Great and Powerful Library sort it out:

```
1 | optimizer_hint(need_locks(UI, NETWORK, DATABASE));  
2 | //snip...
```

Okay, coding that optimizer logic is going to be nasty, but whatever, maybe it's solveable. But what kind of locks does the **promise list of promises** need? Well, it needs the locks that its elements need, of course. Every promise that is composed of other promises, needs to request at least the locks that its children request.

And now what you have is a very slow, very hard to debug, **sequential program**. Because you have, way up top somewhere, a list that contains all the promises. And that list needs all the locks. And nothing else can run.

You might not believe me. I didn't, the first five times I tried to work around the problem. For example, you might say that the list itself needs to acquire no locks, and that only its elements do. And that works right up until you get here:



So your optimizer sees "list promises" 0 and 1, which contain promises [2,3] and [4,5], respectively. 0 and 1 themselves do not need any locks. The optimizer says "Oh good, we can run these both at the same time!" So it starts running 0 and 1, which each run their first element, 2 and 4, respectively.

Which each try to acquire two locks. And deadlock. And so: you really need to acquire all the locks you're going to acquire at the very top of each tree.

Or you may say, "this is because you're executing top-down. Execute bottom-up; start executing the leaf nodes." But now what if it's the root nodes that need to acquire two locks? So you have the same problem; you need to declare the locks up front. All you've done is change whether "up front" means "top" or "bottom"; you've simply flipped the whiteboard upside-down.

And there are a lot of similar "solutions" to this problem, but the cliff notes version is that they all fail under fairly ordinary use cases. For example, you might think of some type of system that acquires one lock at a time, makes a copy of some data, and releases the lock, and then acquires the next lock. And this is a fine theory if A) you have an infinite amount of memory and B) you have an infinite amount of time and C) copy is defined on every object anybody is conceivably working with and D) nobody important changes the object in the meantime and E) you have no particular consistency requirements. There are a lot of **special-case solutions** to this problem, but there's nothing that would work in the **general case**. But maybe we can pick at runtime from a few different choices? Which leads us to:

The atomicity problem

So ideally, what you want, as a programmer, is to register for a promise and to get exactly one result. You don't want "no results" (or your code deadlocks waiting) and you don't want multiple results because maybe that causes weird race conditions.

Simple, right? Well, not so simple:

1. Maybe you're registering before data is available
2. Maybe you're registering after data is available
3. Maybe you're registering after data is available, but we don't have the data anymore due to memory pressure
4. Maybe we loaded the data again after a memory pressure situation, and it is now different than the first time we got it, and the observers got something different than what you're getting now
5. Maybe there was an error and you will never get any data
6. Maybe there was an error but you will still get the data eventually

The problem here is that there is no "one size fits all" solution. Some of the time, you want to be notified of an error and meanwhile keep trying. Other times, you want to give up permanently. Sometimes, if the data changes, you want everybody to know. Other times you don't. And it is hard even to **figure out** if you are in case 4, because we *have removed the data to compare with*. And now you have to negotiate, between the promises and the listeners, what the convention is going to be, at runtime. And so each time you create a promise, you're specifying a ton of metadata with the promise, like:

The size of my result is large, so you might not want to store the result of this computation in memory. Meanwhile, I have side effects, so I can't be re-run. Maybe you can cache a result that's higher up the food chain instead of mine? Good luck with that. Oh, and since I have side effects, you don't want to re-run the code in an error condition, because who knows how much of it got done before the error was raised.

And meanwhile the listener/observer is making demands like this:

I need to paint something to the screen **right now**, so give me an answer or tell me you can't get one quickly. Use the sentinel value NULL to indicate that you couldn't get something in time. Oh, NULL is a legal result for the promise? Okay, I guess I'll just give up myself if you don't return quickly, (can I tell you to cancel at least so we don't hold the locks open too long?) and I'll just ask you again when I paint the next frame. Oh, I can't run the promise again? Bummer. Well just keep it in memory then.

You might think this is some weird contrived case, but this metadata has a **strong likelihood of evolving over time**. Consider the following example, derived from a real-world listener that I tried to write:

Look, I don't want to write error-handling code here. Just figure out how to get me the result of the promise.

Followed quickly by:

Why didn't you get me a value for the promise? (Answer: because an error occurred, and the side-effectful promise can't be re-run, so you can't get a value.) Okay well if you **really want to**, you can return the sentinel value "sorry" to report a "no you're not getting this value" error. But only use it if you **really can't get the value**.

Followed quickly by:

Why the #*\$& am I still not getting a value? (Answer: because the server keeps returning 503 Try Again Later, and you've asked us to keep trying.) No, I didn't mean **try forever**, I meant **try for 10 seconds**.

And so now you have these crazy piecewise definitions like:

```

01  ###
02  def promise:
03      if the listener wants an answer in 0-10ms do X at most three
04          times
05      else if the listener wants an answer in 10ms-1s do Y, which can
06          only run once
07      else do Z
08  ###
09  def listener:
10      if the promise gets us a value inside 0-50ms do A,
11      else if the promise gets us a value inside 50ms-10s do B
12      else if the promise hasn't gotten us a value by now call error
13          handler C with an explanation.

```

Oh Great and Mighty Library, which codepath should we pick here, exactly? And what kind of an error message should we return to C? Do you want a massive debug output that explains that we decided to try codepath X on the promise, it failed twice with two different underlying errors, the third one timed out, and we didn't have time to try Y or Z? Did you want us to just say "ran out of time" or something? Should we maybe lead with codepath Y or Z next time, or try the same thing again?

But it's **worse than that!** Because we have these awful promise lists of promises going on. So now you have a list of promises, where each element in the list **takes a different view about retrying and being notified of updates and so on**. And somebody registers for an update **on the list**. What the heck is supposed to happen now?

Yeah so remember how we were just going to just "describe relationships between values that tell the machine what we want to compute, and the machine *figures out the instruction sequences to make it happen*"? Remember how promises let us "treat the result of the function as a value in a time-independent way"? And where we've arrived is that we need a **full DSL** the **entire purpose of which** is to specify a control flow and timing information. That the programmer **already knows how to write imperatively!** And it's **still not right**.

The solution

So the big problem here is **scope creep**. We started out with an idea about promises being placeholders for values that aren't ready yet, and somehow via a series of what seemed at the time to be perfectly reasonable steps, we ended up with a DSL and a single-threaded program and error messages that nobody can explain. We took a wrong turn somewhere.

What we really should have done is get a lot firmer about rejecting scope creep. Your promises don't fit in memory? Maybe don't use promises for big things then. (Or: you can't register for a promise after it's started.) Your code isn't happy waiting an indefinite period of time for a result? Maybe don't use promises then. You want to deviate from the normal behavior for retrying / error handling? Figure it out yourself. You want promises for lists of promises? Sorry, Resolved (By Design). Your promises deadlock? Sorry, don't use more than one lock. These are situations that we need to **take back to the programmer** and say "Sorry, this is out of scope. Solve these in your application, like you do already."

Of course, if you take this approach, you've given up the dream of solving all the control flow problems, and now you have what **really is** just a different syntax for a callback. But, **the control problems were unsolvable anyway**. So you haven't really lost anything, except your pride and 20,000 lines of code that didn't work.

Now just for the record, I **still like** promises, and I still write the library that uses them. But they are sterile, lifeless, and handcuffed versions of what the promise evangelists are trying to do. And every time I get the urge to extend my library to handle some new corner case, I stop what I am doing, take a deep breath, and slap myself very hard. Bad programmer.

An optimization

It occurs to me at this very late hour, after I have already written and abandoned the entire enterprise, that there was (at *least*) one way to see the problem right from the start. When people say things like:

You can give any acyclic dependency graph to this library and it will optimize the control flow for you.

or

promises provide a way to describe problems using interdependencies between values, like in Excel, so that your tools can correctly optimize the solution for you, instead of you having to figure out control flow for yourself.

What they are really saying is: “Hey kids! Remember that [concurrency problem? Solved it.](#)” Which is crazy, on the level of claiming to have built a perpetual motion machine. If [scheme] could possibly optimize dependencies between values in the general case, then [compiler] could use it to optimize [language]. And either the compiler can, in which case it already does, in which case you have solved nothing that wasn’t already happening at a lower level in the software stack. Or alternately it doesn’t, because the solution you propose doesn’t actually work. If the node “hey, I can solve the concurrency problem” is reachable from your current hypothesis, **something is wrong with the hypothesis**.

Now, of course, we can solve **some** concurrency problems. For example, James produces a promise-based JS loader which as far as I can tell is entirely reasonable. But the trouble starts when you take a solution that works for problems without locks, and that fit in memory, and that don’t spawn thousands of promise objects, and start reasoning about problems that **do all those things**. The former is a small subset of the code that people write, and it is okay to solve the concurrency problem for nice little subsets. What’s not okay is to solve it in general; that breaks the rules. And if you find yourself saying things like “the biggest design mistake committed by Node.js [is] the decision... to prefer callback-based APIs to promise-based ones”, you’re operating under the *general assumption* (assuming, of course, that NodeJS is a general-purpose language). Which **can’t** be right. **Of course** not everything is going to be optimized by a control flow library, that’s a **fundamental** no-no. That should have been my clue right from the start. Unfortunately, I missed it.

Promise hell

And lastly, I want to conclude with a [quote](#) that I stole from James, although we seem to have a different view about what it means:

And when we do hit these problems like callback hell, I’ll tell you a secret: there’s also a coroutine hell and a monad hell and a hell for **any abstraction** you create if you use it enough.

And there’s also a promise hell, and it’s equivalently bad to all the other hells. I write this in the hope that everybody can avoid having to go there themselves.

Want me to build your app / consult for your company / speak at your event? Good news! I’m an [iOS developer for hire](#).

Like this post? Contribute to the coffee fund so I can write more like it. [Donate](#) 0

Comments



1. [Jason Mulligan](#)
Fri 10th May 2013 at 11:20 am

Haters gonna hate.

You clearly haven’t figured out the appropriate place to utilize the pattern, but you nailed the “I don’t like this! rawr!” part common to devs these days.

Congrats.



2. [Andy Matuschak](#)
Wed 10th Jul 2013 at 3:38 pm

This is an excellent case study for API authors, Drew; thank you. I really appreciate all the detailed explanations of the failure modes and the paths you took to get there, rather than the hand-waving hate one typically reads.



3. [Tane](#)
Fri 19th Jul 2013 at 12:53 am

“The war between promises and callbacks is kind of like the war between while loops and for loops.”

I love the unspoken understanding that no-one uses do-while loops. 😊



4. [Lior Bar-On](#)
Sun 21st Jul 2013 at 2:09 am

Thanks for a Great Article!

It does indeed seem Promises are a bit overrated, and it good you nailed down some of its downsides.

I don't feel I'm the right person to do that, but I think some of the issues you raised (e.g. memory consumption) would also happen in naive callback-based implementation.

All in all this doesn't change the overall conclusion, that Promises are not silver bullets.

Lior



5. Marek Sieradzki
Mon 05th Aug 2013 at 2:52 pm

What you're talking about sounds a lot like space leaks in Haskell caused by laziness. It seems that you're explaining it using different words but mean exactly the same thing.



6. Joe
Mon 12th Aug 2013 at 10:55 am

@Tane That just isn't true. I've been quite annoyed by the fact that Python doesn't have do-while loops numerous times. As a matter of fact, I have a Visual Studio window open next to the browser and there is currently a do-while loop on screen. They have their uses.



7. jayprich
Mon 26th Aug 2013 at 11:58 am

Excel's lazy evaluation tree is pretty efficient and parallel. The functions are supposed to be stateless without side-effects, you can break that with VBA of course.

The scattering of the execution path is a core issue for this programming style that can only be solved with better tools, e.g. ability to display the 'trace' for debug.

It seems your complaint is not about a "promise" per-se but that Objective-C and it's libraries do not guarantee a functional style outside Grand Central Dispatch and Blocks (closures).

Do you think iOS 7's multitasking will help address the issues that you say "promises" don't?

It does seem a number of people haven't given up trying this direction of development:

<https://github.com/mogeneration/functionalkit>

<https://github.com/stuarrvine/OCTotallyLazy>

<https://github.com/leuchtetgruen/functional.m>

(OCaml used in an iOS app <http://psellos.com/ocaml/example-app-slide24.html>)

Add comment

Your name*

Your email address* (will not be published)

Your website

Your comment

Notify me of follow-up comments by email.

Notify me of new posts by email.

[Back to top](#) [Back to top](#)

Tags

[app store](#) [arduino](#) [free speech](#) [hardware](#) [HN](#) [incentives](#) [iphone](#) [law](#) [linux](#) [long articles](#) [mips](#) [native apps](#) [notifo](#) [programmers](#) [rants](#) [steve jobs](#) [web apps](#) [wifi](#) [xcode](#)

Subscribe via e-mail

Subscribe via e-mail

Email Address *

