Compressive Strength and Parameter Passing in the Physics of Software

Draft version 1.5

Updates from version 1.4 are in green. Some paragraphs have been readjusted in a different narrative sequence.

Abstract

From the *Physics of Software* perspective, software is a material, reacting to forces according to its own properties and shape. A shape based exclusively on function (or method) calls and parameter (or message) passing is characterized by weak compressive strength, manifested by well-known design and maintenance issues. Known solutions exist within both the functional and object oriented paradigm, although their relationship with compressive strength has not been well understood so far. Some of those solutions have been abused where no compressive strength was required, and therefore prematurely labelled as "anti-patterns". A more precise understanding of forces will lead to more informed design decisions and avoid further pendulum swings.

Introduction

One of the goals of my work on the Physics of Software [1] is to gradually formalize a model of *software as a material*, where non-functional *properties* are defined as *reactions* to *stimuli*, or *forces*.

Today, the idea that software constructs may exhibit anything like *compressive strength* may seem far-fetched, as we normally consider software immaterial.

However, just like we choose and shape physical materials to respond best to contextual forces, so we do for software. We just don't know the forces well enough to recognize and name them properly.

I'll start by illustrating how framing properties of physical materials as reactions to forces provides guidance in their adoption and helps shape them properly, and how this understanding suggests different shapes when different materials are adopted. I'll then move to consider a known phenomenon in software evolution under the perspective of the physics of software, uncovering both a force and a property defined as reaction to such force.

Beams and Arches

The simplest structure that can bridge two points is a beam, as represented in fig.1. The beam is loaded with its own weight, plus any additional weight (structures, vehicles, water) that it's being designed to carry.

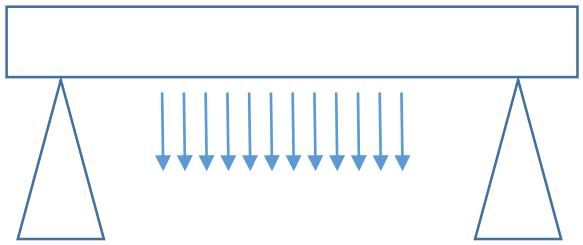

Fig. 1

That type of loading results in the beam bending, as represented (greatly exaggerated for emphasis) in fig. 2.

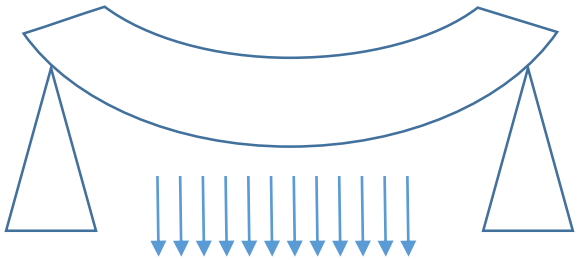

Fig. 2

Bending, in turn, exerts compression on the material on top of the beam and tension on the material on the bottom (fig. 3).

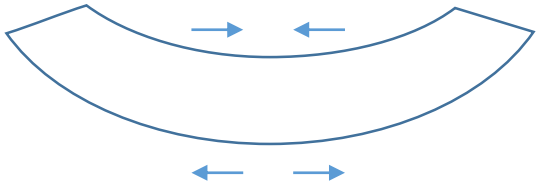

Fig. 3

Construction materials easily available to early civilizations (stone, marble, wood) tend to have good *compressive strength* (ability to sustain compression) and poor *tensile strength* (ability to sustain tension). With reference to fig. 3, they would "break on the bottom", where tensile strength is required. Therefore, a bridging structure shaped as a simple beam was not a good fit for the material, and when adopted required a narrow distance between columns.

The roman arch (fig. 4) was conceived exactly to transfer most of the load in form of compression. A *change in shape* was therefore required due to the low tensile strength of the adopted materials.

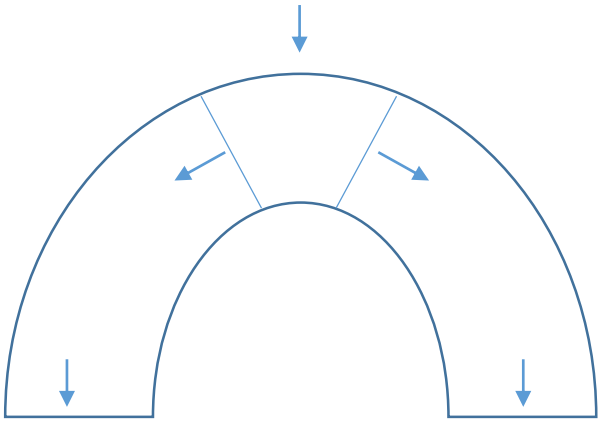

Fig. 4

When materials with high tensile strength (mostly steel) became available, the beam (especially the I-beam) in turn became a better fit; other shapes, like suspension bridges, also became more popular.

Of course, given enough pressure, any material will fail. Materials will fail under compression in different ways; some will fracture from top to bottom as the timber block under test in fig. 5:

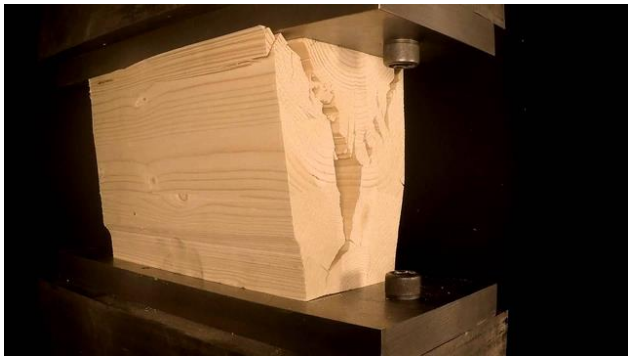

Fig. 5

It's ok to fail

Before we move to software, I think it's important to reflect on the objective and dispassionate perspective of science. Having low tensile strength is not considered a shame; in fact, marble is still routinely used in many roles where tensile strength is not required. Conversely, there are also no attempts to construe tensile strength as irrelevant or overshadowed by other "more important" properties. Instead, we recognize reality and learn to shape our material so that compression is carried more than tension when needed. Of course, materials may have other properties more desirable, in a specific context, than tensile strength.

Parameter Passing

A good theory should be informed by practice, observation and experiments, so let's start from a realistic example – parsing a "complex" file format.

Let's say that we need to parse a binary file organized into a header, containing a number of meta-data about the file itself, followed by the actual contents, organized in a number of sections having variable length, possibly with nested sub-sections, down to domain-specific values (coordinates, or colors, or quantities) mapped to the usual strings, integers and floats[1].

For sake of simplicity, we can simply ignore that we're reading a file and consider the similar problem of parsing data from an in-memory array, removing the need for I/O and mapping more directly into a functional world.

It would seem reasonable to organize the parser using a functional decomposition that mirrors the information structure / grammar of the data being parsed (fig. 6).

That's relatively straightforward, until you realize that one of the parameters in the headers is describing the binary format of integers in the contents section – big endian or little endian[2].

[1] Similar file formats are quite common in the industry. Images, GPS / fitness data, lab measurements, etc. are often structured in files with a similar information architecture.

[2] I am setting this as a design problem, as all this information is potentially available upfront. In practice, in a number of real-world cases this would manifest as a maintenance problem, as information is gradually discovered or changes are required due to new requirements. Incremental discovery only amplifies the issues discussed in what follows.

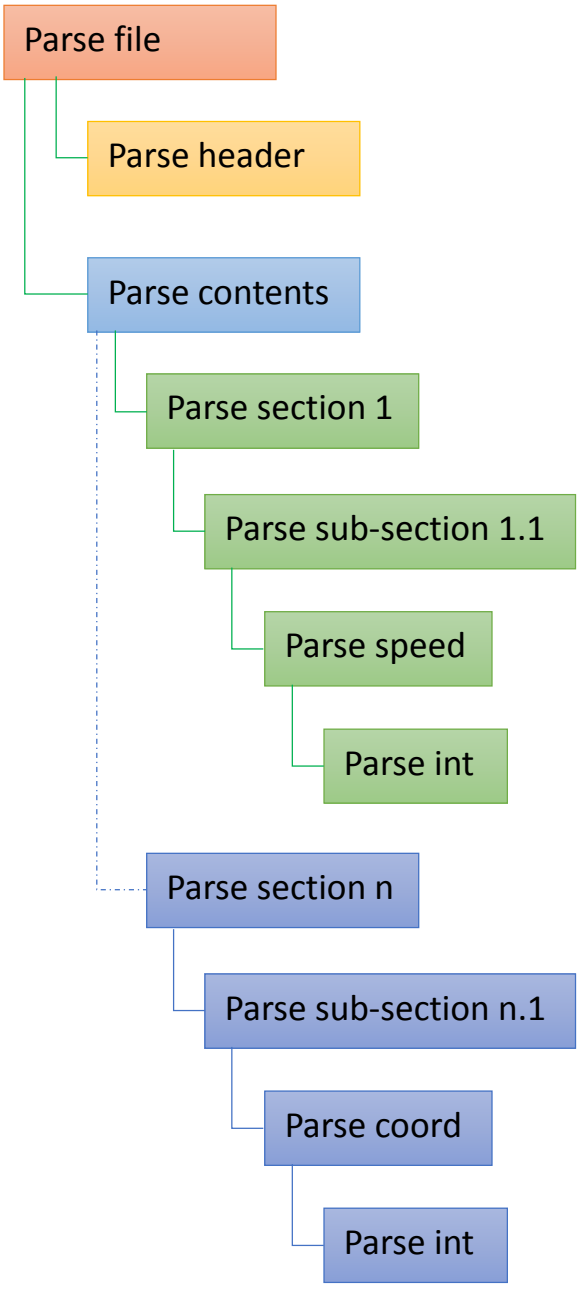

Fig. 6

That can be easily solved by passing that information as a parameter, from the "parse file" function (which got it from "parse header") down to the "parse contents" and further down to all the "parse int" occurrences.

Of course, by doing so we need to change the entire chain of calls from the top level – where information is known – to the bottom level – where information is needed. We need to break the existing interface of every involved function. We need to crack every function open and change its implementation to forward the new parameter down the chain.

All the intermediate functions are "fractured" by this change, top to bottom, just like the timber log in fig. 5, without any *local* need or benefit – those functions are concerned with parsing complex structures and don't need to know the integer format.

The fracturing, of course, does not stop there. As you progress, you learn that strings can be marked as UTF-8 or as ANSI in the header. You need to pass that information down. Another crack is opening. Colors might be represented as RGB, ARGB, etc. Another fracture.

At that point, many programmers will move from a number of small fractures to a single large fracture, passing an algebraic data type that may more or less correspond to the entire header, so that individual functions can "pick" the parts they need. At least this will prevent further fractures: we'll simply make the single fracture larger on demand, by adding fields to that single type as required.

This is ok *syntactically* - semantically we still have a large chunk of information going through a number of functions just to be passed downstream. Depending on the organization of the different sections and sub-sections, the richness of the header, etc., that lump of information will probably lack cohesion or, in the more precise formulation within the Physics of Software, fields won't be Read/Read entangled [2], [3] either in the artifact space and

in the run-time space, and therefore should not be kept together if we aim for locality of action[3]; however, we may still opt to do so because it looks like the most convenient option. Worst case: we pass the entire set of configuration data down to "parse int" and it will pick the only parameter that's actually needed (endianness).

While the example above is built around factual knowledge (data) needed by a bottom layer and known only at the top layer, an identical scenario would present itself if the knowledge was operational (functions). As you discover new operational knowledge to be passed downstream, the intermediate layer will fracture just like when passing data.

There are, of course, known solutions to this problem, but not within function composition and parameter passing. Composition, often touted as the quintessential tool in the functional arsenal, does not help at all here. The only place where we know enough to "compose" the right function is the top level; the place where we may want to use the "composed" function is the bottom level. "Parse int" was a direct function call in the initial implementation, so we still have to fracture all the functions from top to bottom to pass it as a parameter.

In the contemporary narrative of computing, this is where we normally split between the apologetics – trying to prove that parameter passing is still the best thing on earth – and the harsh critics, quick to define parameter passing as a dangerous anti-pattern.

[3] The shortcomings of passing a large type are probably more self-evident if you consider some of the layers as **modules** that have to carry downstream data they have no use for – and unstable data inasmuch as we keep adding fields.

The perspective of science is different, and in itself quite simple: if indeed we are cracking that stack of functions top to bottom, there must be some force at play, and we should identify the force, and then characterize our materials according to their reaction to that force.

Information differential

Moving away from the concrete example above, we can now consider a general case where a function f_1 owns some information K that is required by a function f_n, which is indirectly called by f_1 through a series of function calls to f_2 … f_{n-1}, where f_2 … f_{n-1} have no use for K.

This can be represented as in fig. 7:

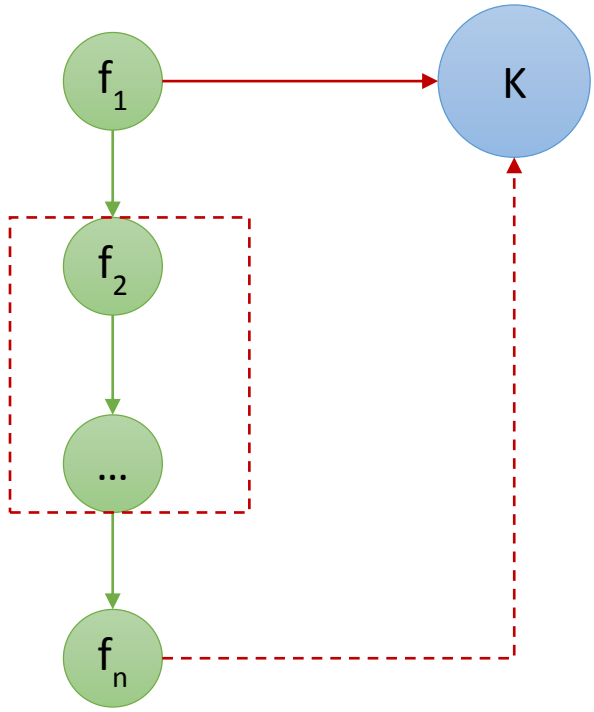

Fig. 7

An alternative representation, more aligned with the following reasoning (albeit less aligned with traditional representations of software artifacts) is provided in fig. 8, where function f_1 is characterized by an "excess" of information K, f_n by a lack of information K, while f_2 … f_{n-1} would ideally be oblivious to / isolated from K.

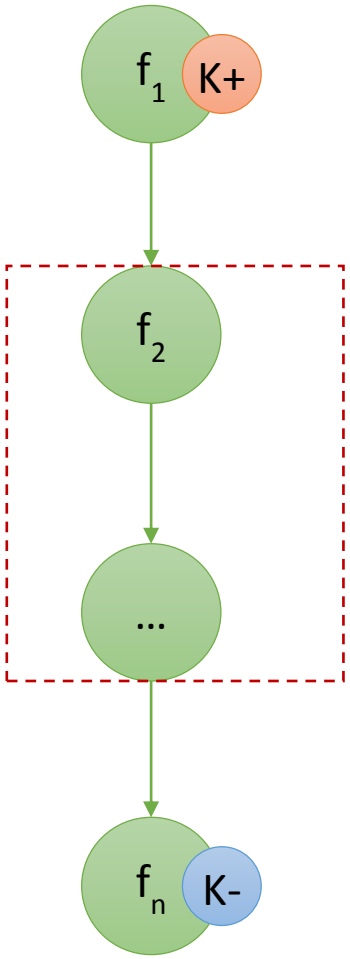

Fig. 8

The unbalance (differential) in information contents is creating a **force of attraction** (see fig. 9) between f_1 and f_n, even though they are "distant" in the logical and physical decomposition of our code. That force has to be balanced through the choice of an appropriate material and shape, as **in a working system the information differential must be zero along all paths.**

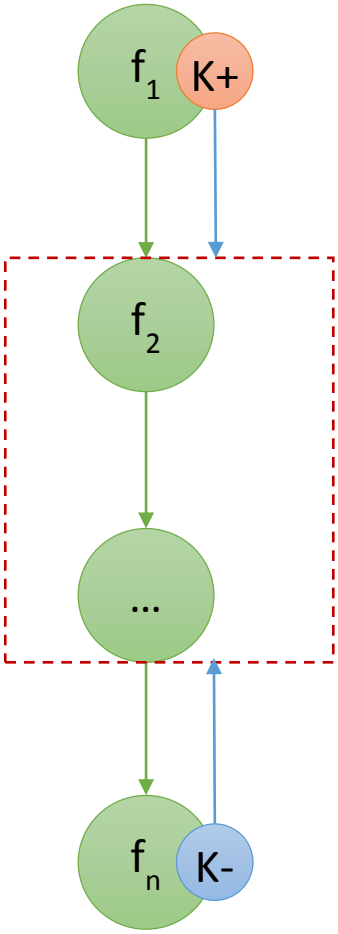

Fig. 9

As we'll see, different shapes and materials will respond differently. A common approach would be to choose parameter passing. Parameter passing, as already exemplified, will cause all the intermediate functions f_2 … f_{n-1} to "break" in their interface and implementation, so that information K can get through, as shown in fig 10 (where K is in gray, as it's no longer unbalanced).

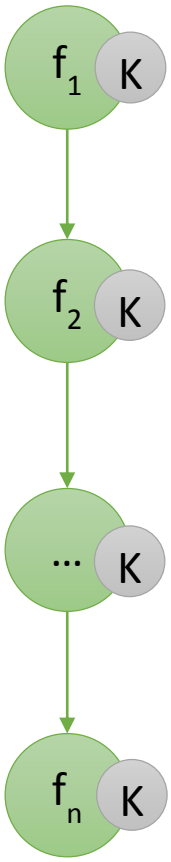

Fig. 10

This is the software equivalent of structural failure, where intermediate layers are now tainted by information they have no use for.

A first-cut quantitative definition of information differential

Although I'm wary of proposing quantitative models too early, it seems reasonable at this stage to define the information differential Δ_K between f_1 and f_n as a vector of magnitude K and direction $f_1 \rightarrow f_n$. This, however, requires that we define the magnitude of information K and a notion of distance for the artifact space (so that it could be properly characterized as a metric space). While it's too early for the latter, quantifying the magnitude of K is already within reach. The only real challenge is to formulate a robust definition under (possibly tricky) code transformations.

Here the method is extremely important, as within the perspective of the Physics of Software **we want to quantify notions in the artifact space by observing what happens to the artifacts**, unlike many metrics that tend to define and quantify notions that are easy to measure but are not direct mappings of observable phenomena.

Within the methodological framework of the *representational theory of measurement [4]*, we should start with an understanding of the empirical structure we want to quantify, and of the relations that must hold within that empirical structure, and then define our measurement in a way that preserves the relations in the empirical relational system.

Now, considering the simple case of parameter passing, empirically the magnitude of K should be proportional to the number of individual fractures caused by K. Therefore, adding more knowledge would increase K in direct proportion, which seems consistent with the empirical experience.

However, if the fracture is not individual (as when we lump together individual values in a larger value using an algebraic data type) the magnitude of K should not change, even though we have only one fracture.

Still, when the unit of knowledge required by f_n is indeed composite in nature (for instance an array of integers that will be used uniformly, e.g. to calculate an average) the contribution to the magnitude of K must be unitary, as empirically it is rather obvious that the information

differential won't change if we make the array bigger (case in point: if we already pass the array, Δ_K is zero and will stay zero if we increase the size of the array).

However, if we try to play the system and pass (e.g.) two unrelated integers as a single array, the magnitude of K should be the same as when two independent values are passed.

With that empirical relational system in mind, it's rather obvious that:

- The magnitude of K has nothing to do with the amount of bytes transferred at run-time from f_1 to f_n.

- The magnitude of K depends on the usage of K within f_n (uniform or not).

Conscious that I'm still introducing more notions to be formally defined, I can still provide the following first-cut definition of information differential:

Definition 1: an information particle[4] known to f_1, needed by f_n, and having distinct identity within f_n is called an **unbalanced** particle between f_1 and f_n.

Definition 2: the **magnitude** of the information differential Δ_K between f_1 and f_n is the cardinality of the set of unbalanced particles between f_1 and f_n.

The notion of identity in the artifact space will be defined in a subsequent paper. Informally, you can think of an information particle having identity within a context if it has a distinct name within that context. Composite distinct names like a[0] are still distinct names. This accounts for non-uniform usage of values passed within arrays.

Note that this definition accounts – using the informal terminology adopted in the previous paragraphs – for both individual, single-parameter fractures and fractures made "larger" by adding fields / dimensions to a single type.

In fact, if we consider the case where information K is being transferred from f_1 to f_n as a set of parameters P_1 … P_m, the following must hold, independently of the number and type of the parameters:

Equation 1:

$$\sum_{i=0}^{m} |U(P_i)| = K$$

Where $U(P_i)$ is the set of unbalanced particles carried by P_i.

[4] I haven't introduced this term formally yet. Informally, in this context you can think of it as a value you can pass.

Compression and Compressive Strength

When a stack of bricks break top to bottom, it's usually due to a compressive force being applied (fig. 11) until the compressive strength of the material is reached (see also fig. 5 again for a real-world example).

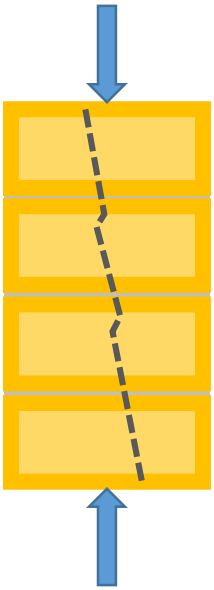

Fig. 11

This is reminiscent of what happens to artifacts under an information differential, when the shape is vertical stacking with parameter passing.

That, of course, is not the only physical phenomenon that is known to cause fractures within a material. For instance, a dielectric will fail (and fracture) once the difference of electrical potential applied gets above its dielectric strength.

While these parallels might be seen as random metaphors, it pays to understand that within physics and engineering there is a well-developed theory of equivalence between mechanical, electrical, and other energy domains (see for instance [5]) dating back to Maxwell. Within that theory, the mechanical equivalent for voltage is a force, and in this perspective choosing one analogy or the other is mostly a matter of convenience and communication. I think the mechanical analogy is easier for people to grasp intuitively, and therefore I'll stick to that in what follows.

By extending the analogy to the information (artifact) space, we can consider information differential as a force. When there is an information differential Δ_K between f_1 and f_n, all intermediate function[s] f_2 … f_{n-1} will be subject to a form of **compression** (with magnitude Δ_K). When intermediate functions break, we can further extend the analogy and conclude that we have reached their **compressive strength**[5].

Parameter Passing reconsidered

As stated above, when the only shape available is one based on function calls and parameter passing, a form of **compression** is applied to the intermediate functions / layers / modules (see also fig. 10 again) under information differential.

While this is too an early stage to quantitatively characterize the *compressive strength* of software materials[6], it is easy to *qualitatively* conclude that materials and shapes based solely

[5] It is worth highlighting that the choice of compression and compressive strength is meant to provide an immediate, intuitive grasp. The underlying notion of *information differential* does not need to be grounded in physical analogies.

[6] A first step toward a model based on an ordinal scale is introduced at the end of this paper.

on function calls and **explicit** parameter passing have **zero (or no) compressive strength**, as any magnitude of compression will cause a fracture.

Again, from a scientific perspective this should not be construed as a form of blind criticism. Yes, explicit parameter passing, which at the time in which I'm writing these notes is being idealized to the perfect way to transfer information, is evidently not perfect. That does not mean it does not have other benefits; for instance, it is a very *explicit* style, which in the physics of software can be formulated as a *symmetry between R/R entanglement in the run-time space and in the artifact space*. However, pretending that there is no issue with compressive strength would be ignoring reality.

As an aside: this issue has little to do with the common (at least at the time I'm writing these notes – hopefully this too shall pass) debate on functions vs. objects. Objects + message passing would suffer the same consequences if subject to the same constraints, that is, no side effects, no global state, explicit message passing.

Solutions with traditional functional materials

The issues described above are, of course, well known in the professional programming community. They're not formulated in terms of information differential or compressive strength, and are probably not commonly discussed by "evangelists"[7], but this is more a sign of the times than a sign of irrelevance.

Hints to the problem and to possible solutions appear in literature, for instance in *Real World Haskell* [6], chapter 10 and 18. Again a sign of the times is that these issues tend to be discussed in social forums more than in books or formal papers. Examples from *stackoverflow* and *reddit* discussions are in [7], [8], [9], which also discuss a common solution: monads and monad transformers. The same approach is also briefly discussed in [10] under "Relaxed Layers – Monad Transformers".

In short, readers familiar with FP have already recognized the opportunity to save the intermediate layers from forwarding parameters by moving from functional composition to monadic composition. A common way to send information "at a distance" is through the Reader monad, a.k.a. the Environment monad.

In practice, going back to the original example of parsing a file, a single monad won't help much as we have to deal with a number of issues while parsing:

- Optional results in case of failure.
- Variable-length structures require that we keep track of how much data we have already parsed.
- Diagnostics must be produced in case of errors.
- Etc.

A relatively common approach therefore relies on stacking monad transformers, as opposed to creating a non-reusable uber-monad. Either way, function composition and parameter passing have to be replaced with a different material / shape, with higher compressive

[7] It's hard to say if the peak of unprofessionalism in computing is reached when we talk about evangelists and prophets or ninjas and rock stars.

strength. This, of course, may have other consequences, as much of the "simplicity" of function calls and explicit parameter passing is lost.

Solutions with non-functional materials

Faced with the same problem, but freed from the limitations (and of course devoid of the benefits) of referential transparency, the OO community has explored, over time, different alternatives, mostly based on the notion of a shared state, not necessarily mutable, but largely implicit.

On a short distance, *the class itself acts as an implicit context*. All member functions implicitly share all the data members. Note that I'm saying "short distance" here on the assumption that classes are kept small. In fact, it would be interesting to analyze the common emergence of large classes in the average codebase as a relatively cheap way to obtain some compressive strength[8].

Of course, the same notion applies to nested functions, sharing the outer function[s] context. On large distances, a number of solutions have been adopted over time:

- Dependency injection via IoC containers.
- Service Locators, Brokers, Blackboards. In simple cases, just a Singleton.
- Contextual data as Thread Local Storage.
- Etc.

[8] As opposed to the common approach of chastising programmers who create large artifacts, understanding why they do it would allow to create better materials. I too am guilty of simply dismissing long function / classes as "bad practices" or "laziness", perhaps too quickly.

Each of these solutions has been abused, to the point where – in the classical swing of underdeveloped disciplines – they have been labeled as evil, anti-patterns, etc.

Language-level solutions

Solutions to the issue of compressive strength have been sought both within and outside mainstream paradigms. Both Scala and Haskell (the latter through a type system extension) support *implicit parameters*. Roughly speaking, in both cases functions signature will be fractured to preserve static type checking, but functions body won't need changes to propagate parameters downstream.

To make an analogy with fig. 7, this could be represented as in fig. 12.

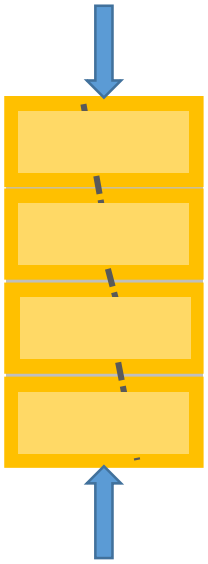

Fig. 12

Interestingly, this approach is often frown upon and labelled as a dangerous hack, without any serious attempt to identify contexts in which it could be useful or ways to make it work better.

Dynamic scoping is an old form of scoping resolution, rarely adopted in modern languages, but relevant in this context.

Unlike implicit parameters, dynamically scoped variables do not appear in the function signature.

Therefore, strictly speaking, dynamic scoping offers higher compressive strength than statically scoped implicit parameters (the signature won't "crack" when a new parameter is required). As usual, they have other consequences, both in terms of human understanding and in the practical impossibility of static checking.

Interestingly, while looking for a good bibliographical entry on dynamic scoping, an old paper [11] from Guy Steele and Gerald Sussman resurfaced in my library. There, the authors explain the practical usage of dynamic scoping in a context quite similar to the one I've been describing above (see page 43, "dynamic scoping as a state-decomposition discipline"). This reinforces the idea that lacking a language- and paradigm-independent characterization and analysis of compressive strength, generations are bound to rediscover the issues, try solutions, stumble into problems, swing back to overly restrictive and oversimplified styles, and so forth, with little global advancement in the discipline.

Among non-mainstream paradigms, both Adaptive Programming (see for instance [12], Chapter 10) and Context-Oriented Programming [13] have tried to deal with the issue.

In Context-Oriented programming, the problem is often reframed as *function selection*, so instead of passing an endianness parameter we reframe as choosing between ParseIntBigEndian vs. ParseIntLittleEndian based on contextual information. Adaptive Programming frames the problem as a *transport problem*, and somewhat subsumes it within *structure-shy navigation*.

Both have probably something to teach, and indeed the separation into "paradigms" (which according to Kuhn brings an issue of incommensurability) is probably less productive than a more engineering-like separation into materials, with much less intellectual and emotional attachment and more natural compatibility.

And yet, bridges fell

It is interesting to get back to construction to gain some perspective.

Stone was (intuitively) understood as having poor tensile strength and a different structure was sought (roman arch). More ambitious artifacts, like wider bridges, were then possible.

However, over time bridges fell; for a long list of known cases of failures in the history of bridges see [14]. Still, the bridge was not declared to be an anti-pattern, neither were the arch nor stone labeled as evil. We gradually understood limitations, formalized forces, improved constructions techniques, explored new shapes and new materials.

As materials with high tensile strength emerged (steel, shaped in beams and cables) new opportunities arose to build different bridges. This did not replace arches entirely; we understand materials well enough to appreciate

their difference. Stone, for instance, is readily available in places where bringing steel would be impractical.

Perhaps surprisingly, even with new materials and shapes bridges kept falling (see [14] again for many cases where steel was adopted), due to wrong calculations, improper construction, unexpected events. Failures are openly discussed and seen as opportunity for improvements.

Modern bridges tend to have complex shapes ("architectures"). Their complex shape might be inspired by creativity and to some degree by aesthetics, but is supported and justified by structural engineering. We can easily contrast this with the recurring obsession with *structural simplicity* in computing, where many "experts" are basically suggesting that using a beam in every single case is ok ("just pass parameters"), mostly as a reaction to having seen the like of a suspension bridge adopted without any real need / benefit. A clear sign of the immaturity of our field is the lack of a good theory supporting informed decisions; instead we're still trapped in a pre-scientific world of "principles" sprinkled with the occasional dogma.

Going deeper

Just like IoC containers have been abused where passing a parameter would have worked just fine, monads and monad transformers will be abused, it's just a matter of time. See [7] again for an increasing awareness of these issue in the Haskell community. Implicit parameters have already been removed from the table in many conversations, dismissed as "evil".

It is my hope that moving away from the commonplace advocacy of languages and paradigms, and toward a deeper understanding of software forces, materials and shapes will provide a long-needed antidote to the pop-culture cycle of hype and despair we keep repeating.

Recognizing the issue of compression and compressive strength is just **one step**. A precise model of how monads, monad transformers, IoC containers, service locators, etc. are altering the force field would offer a much better opportunity for discussion and adoption, free from the oversimplification of "just pass parameters" and from the equally blind oversimplification of "xyz is an evil anti-pattern", hopefully providing the professional programmer with more objective foundations and enabling more informed design decisions.

The relationship between compressive strength and other properties is also worth investigating. Providing an implicit context can easily increase the coefficient of friction as defined in [15], making code harder to move.

Over time, thinking in terms of materials and forces could also help to conceive **new** solutions, either by borrowing from niche paradigms or by engineering a material / shape on purpose.

Quantitative model of compressive strength

While I consider a good qualitative model important, a quantitative model is usually more immediately useful, provided that we understand what we're trying to quantify (the empirical structure, see [4] again) well enough.

We also need to remember that quantification does not necessarily mean assigning numbers; in fact, there are many practical cases where only an ordinal scale can be provided (like "hazard

level"). In fact, given the discrete nature of software, it's unlikely that we'll ever be able to have an interval or ratio scale for every quantity.

Coming up with an ordinal scale is also a way to test our collective understanding and agreement. We can start with a set (not yet a measurement scale) describing the reactions of intermediate layers (f_2 … f_{n-1}) subject to information differential:

{ nothing breaks, only interface breaks, only implementation breaks, both interface and implementation breaks }

To be thorough, we should also consider the case where we do not create a new fracture, but we make an existing fracture "larger" by adding fields / dimensions to an existing parameter.

In practice, at this stage I haven't found any evidence of a case where the implementation breaks under information differential, but the interface stays the same. It's even hard to conceive how that configuration may work. Along the same lines, when we make existing fractures "larger", there is no syntactic change to either the interface or implementation. It's a change to a type mentioned in the interface. That would suggest that we revise the set above as:

{ nothing breaks, only parameter types break, only interface breaks, both interface and implementation breaks }

Now the task of coming up with an ordinal scale is reduced to that of finding a total order in such set, so that empirically we consider α worse than β if α < β, and then we can call that ordinal scale "compressive strength". This would get more complicated if we had kept "only implementation breaks" in such set, but based on the above, I suppose we would all empirically agree that the order, if it indeed exists, must be a linearization of the poset in fig. 13:

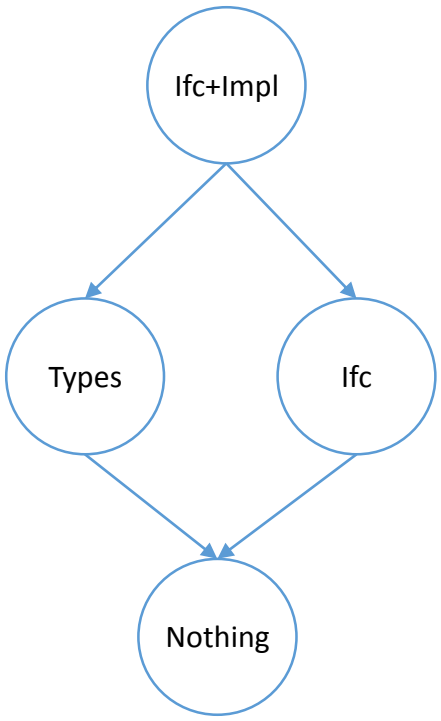

Fig. 13

That is, it's rather obvious that breaking nothing is better than breaking a little and that breaking all is worse than breaking a little. It's not necessarily obvious, and a possible cause of disagreement, whether or not creating a large fracture is better than breaking the interface by creating another explicit fracture (parameter).

At this stage, I am inclined to resolve the issue by making it, unfortunately, more complicated, bringing in the notion of R/R entanglement in the run-time space. Since this work would only make sense once a circuit model of information differential has been worked out (see next paragraph), which could indeed add more cases to the set anyway, I will leave the issue open in the present draft.

Further developments

- You may recognize a similar, symmetrical issue when returning results (up) from functions, as opposed to passing parameters (down). Here again, it may happen that intermediate layers have to propagate some results back to the top-level functions, results that they don't actually need but have to be aware of. This is particularly common for detailed diagnostics. Much of the same reasoning as with parameters applies. For instance, exceptions were initially seen as a mechanism to increase compressive strength, and partially because of poor choices by language designers, they failed in many languages (see the large number of try/finally in intermediate layers) and are now discredited in favor of, unsurprisingly, monads.

A quantification of the information differential can be provided here as well, reversing the role of f_1 and f_n while quantifying magnitude, but keeping the direction as $f_1 \rightarrow f_n$. This works well until we consider mutable in/out parameters, which should be counted only once. That's one of the reason why this is still considered a first-cut quantitative model.

- This paper has considered only the simplest context, where we have an information differential between two single points (f_1 and f_n). As in electrical engineering, it is however important to characterize more complex circuits (call graphs). This step (ideally completed together with a study of return values) should also provide enough details to understand whether or not the parallel with compression holds, or if it's better to move past this stage, adopt only "information differential" as a term, and introduce a new term for what I have now called "compressive strength".

- The definition of information differential relies on terms like information particle and identity in the artifact space, which should be formalized as well. It would also be worth explaining in more details (with examples) why the definition is based on the callee context, not the caller's. This would also be a chance to explain how some "syntactic sugar" that we tend to consider a mere convenience (variable number of parameters) ends up revealing the actual information differential (usually 1, as variable arguments are normally treated uniformly) compared with alternative syntax based on overloading and fixed parameter set.

- It has been proposed that this paper is useless as I do not yet provide either a quantification of compressive strength or a precise characterization of all the aforementioned solutions (IoCs, Monads, etc.) in term of compressive strength. This kind of criticism is again a sign of immaturity of our field: while in more established disciplines there are entire books devoted to a single property for a small family of materials (like "elasticity in rubber"), we expect a seminal paper to have everything figured out and quantified in the space of a few pages. In early stages of theory formation, I consider qualitative characterization of phenomena a useful and important step toward a quantitative model. We have seen many examples where scholars have proposed more or less sophisticated equations (just think of the literature on coupling and cohesion) that nobody is using in practice, relying instead on qualitative, intuitive understanding. A complete characterization of solutions in terms of compressive strength *alone* would be useless, because higher compressive strength does not automatically equate with "better". Other balancing effects need to be formalized, like

notions that we only appreciate informally (like delocalized plans) plus other contributing factors like reentrancy / thread safety, not to mention the effect of distance[9], and until the impact of those notions is clearly understood from the qualitative perspective I will restrain from systematic comparison of alternative solutions.

- Two reviewers asked me if we also have a form of tension in software. The short answer is yes, there is a force that I'm currently inclined to classify as tension, but would be out of scope here.

Feedback

Constructive feedback is welcome. For short messages consider twitter (@CarloPescio) but for a more reasoned conversation I've set up a message board, and there is a specific topic for this paper, so consider posting there. Thanks.

Acknowledgments

The failing timber block in fig. 5 is extracted from [16], which is released under a Creative Commons Attribution - NonCommercial 3.0 Unported License.

Thanks to Paolo Bernardi, Claudio Brogliato, Egon Elbre, Michel Mazumder, Daniele Pallastrelli, Michelangelo Riccobene, Marius Schultchen, Giorgio Sironi for their comments and suggestions on early drafts of this paper. Of course, any residual error / lunacy is entirely mine.

[9] When n is small, breaking interface and implementation might be just ok in exchange for the explicit nature of parameter passing.

References

[1] *http://physicsofsoftware.com*

[2] Carlo Pescio, *Notes on Software Design, Chapter 13: On Change*, 2011.

[3] Carlo Pescio, *Notes on Software Design, Chapter 15: Run-Time Entanglement*, 2011.

[4] Fred S. Roberts, *Encyclopedia of Mathematics and its Applications Vol. 7: Measurement Theory with Applications to Decision making, Utility, and the Social Sciences*, Cambridge University Press, 1985.

[5] Wikipedia, *Mechanical-electrical analogies.*

[6] Bryan O'Sullivan, John Goerzen, Donald Bruce Stewart, *Real World Haskell*, O'Reilly Media, 2008.

[7] https://www.reddit.com/r/haskell/comments/4c533b/tips_for_organizing_monadic_code_better/

[8] http://stackoverflow.com/questions/12968351/monad-transformers-vs-passing-parameters-to-functions

[9] http://stackoverflow.com/a/3083909

[10] Alejandro Serrano, *Teaching Software Architecture Using Haskell*, International Workshop on Trends in Functional Programming in Education, 2014.

[11] Guy Lewis Steele, Gerald Jay Sussman, *The Art of the Interpreter or, the Modularity Complex*

(Parts Zero, One, and Two), MIT AI Lab. AI Lab Memo AIM-453, May 1978.

[12] Karl Lieberherr, *Adaptive Object-Oriented Software, The Demeter Method* , 1996.

[13] Robert Hirschfeld, Pascal Costanza, Oscar Nierstrasz, Context-oriented Programming, The Journal of Object Technology, Volume 7, no. 3, March 2008.

[14] Wikipedia, *List of bridge failures*

[15] Carlo Pescio, *Software Design and the Physics of Software*, DDD EU 2016, Bruxelles.

[16] Christian Malaga - Chuquitaype, *Compression failure of a timber block perpendicular to the grain: Materials Lab on-line*, Imperial College, London.

About the author

As I'm writing these notes, I've been programming for 38 years, using different languages, paradigms, technologies. I've been an author and an occasional speaker. Over time I've learnt a few things, within and outside the academia. I'm slowly building a well-founded and hopefully useful theory of software design. For a longer version: *physicsofsoftware.com/me*