On compositionality

Jules Hedges

2016

This short essay is copied (with some small modifications and additional comments) from section 0.4 of my Ph.D. thesis, *Towards compositional game theory*. It is loosely based on a talk I gave at Logic for Social Behaviour 2016 in Zürich.

The term *compositionality* is commonplace in computer science, but is not well-known in other subjects. Compositionality is the principle that a system should be designed by composing together smaller subsystems, and reasoning about the system should be done recursively on its structure. When I thought more deeply, however, I realised that there is more to this principle than first meets the eye, and even a computer scientist may not be aware of its nuances.

It is worthwhile to spend some time thinking about various natural and artificial systems, and the extent to which they are compositional. To begin with, it is well known that most programming languages are compositional. The behaviour of atomic[1] statements in an imperative language, such as variable assignments and IO actions, is understood. Functions are written by combining atomic statements using constructs such as sequencing (the 'semicolon' in C-like syntax), conditionals and loops, and the behaviour of the whole is understood in terms of the behaviour of the parts together with the ways in which they are combined. This scales sufficiently well that a team of programmers can broadly understand the behaviour of a program consisting of hundreds of millions of individual atomic statements.

When the software industry began software was unstructured, with no intermediate concepts between atomic statements and the entire program, and much of its history has been the creation of finer intermediate concepts: code blocks, functions, classes, modules. Compositionality is not all-nor-nothing, but is slowly increased over time; nor is it entirely well-defined, with many tradeoffs and heated debates in the design and use of different language features. Even with a modern well-designed language it is possible to write bad code which cannot be easily decomposed; and even though there are many design patterns and best practice guidelines, good software design is ultimately an art.

Going beyond software, consider a physical system designed by human engineers, such as an oil refinery. An individual component, such as a pump or a section of pipe, may have a huge amount of engineering built into it, with detailed knowledge of its behaviour in a wide variety of physical situations. It is then possible to connect these components together and reuse knowledge about the components to reason about the whole system. As in software, each component has an 'interface', which is a high level understanding of its behaviour, with unnecessary details being intentionally forgotten.

[1]The term 'atomic' is used naively here, and does not refer to concurrency.

As a third example, an organisation made of human beings, such as a company or university, is also built in a compositional way, demonstrating that engineering is not a requirement. It is possible to understand the behaviour of a department without knowing the details of how the behaviour is implemented internally. For example, a software engineer can use a computer without knowing the exact process through which the electricity bill is paid, and will probably not even be aware if the electricity provider changes. This is another example of reasoning via an interface.

Clearly interfaces are a crucial aspect of compositionality, and I suspect that interfaces are in fact synonymous with compositionality. That is, compositionality is not just the ability to compose objects, but the ability to work with an object *after intentionally forgetting how it was built.* The part that is remembered is the 'interface', which may be a type, or a contract, or some other high-level description. The crucial property of interfaces is that their complexity stays roughly constant as systems get larger. In software, for example, an interface can be used without knowing whether it represents an atomic object, or a module containing millions of lines of code whose implementation is distributed over a large physical network.

For examples of non-compositional systems, we look to nature. Generally speaking, the reductionist methodology of science has difficulty with biology, where an understanding of one scale often does not translate to an understanding on a larger scale.[2] For example, the behaviour of neurons is well-understood, but groups of neurons are not. Similarly in genetics, individual genes can interact in complex ways that block understanding of genomes at a larger scale.

Such behaviour is not confined to biology, though. It is also present in economics: two well-understood markets can interact in complex and unexpected ways. Consider a simple but already important example from game theory. The behaviour of an individual player is fully understood: they choose in a way that maximises their utility. Put two such players together, however, and there are already problems with equilibrium selection, where the actual physical behaviour of the system is very hard to predict.

More generally, I claim that *the opposite of compositionality is emergent effects.* The common definition of emergence is a system being 'more than the sum of its parts', and so it is easy to see that such a system cannot be understood only in terms of its parts, i.e. it is not compositional. Moreover I claim that *non-compositionality is a barrier to scientific understanding*, because it breaks the reductionist methodology of always dividing a system into smaller components and translating explanations into lower levels.

More specifically, I claim that compositionality is strictly necessary for working at scale. In a non-compositional setting, a technique for a solving a problem may be of *no use whatsoever* for solving the problem one order of magnitude larger. To demonstrate that this worst case scenario can actually happen, consider the theory of differential equations: a technique that is known to be effective for some class of equations will usually be of no use for equations removed from that class by even a small modification. In some sense, differential equations is the ultimate non-compositional theory.

Of course emergent phenomena do exist, and so the challenge is not to avoid

[2]I learned of this point of view from Michael Hauhs, at the 2015 Dagstuhl seminar *Coalgebraic semantics of reflexive economics.* I believe it is originally due to Robert Rosen.

them but to *control* them. In some cases, such as differential equations, this is simply impossible due to the nature of what is being studied. Using open games it is possible to control emergent effects in game theory, although it is far from obvious how to do it. A powerful strategy that is used by open games is *continuation passing style*, in which we expand our model of an object to include not only its behaviour in isolation, but also its behaviour in the presence of arbitrary environments. Thus an emergent behaviour of a compound system was already present in the behaviour of each individual component, when specialised to an environment that contains the other components.

As a final thought, I claim that *compositionality is extremely delicate*, and that it is so powerful that it is worth going to extreme lengths to achieve it. In programming languages, compositionality is reduced by such plausible-looking language features as goto statements, mutable global state, inheritance in object-oriented programming, and type classes in Haskell. The demands placed on game theory are extremely strong[3]: seeing a game as something fundamentally different to a component of a game such as a player or outcome function breaks compositionality; so does seeing a player as something fundamentally different to an aggregate of players; so does seeing a player as something fundamentally different to an outcome function. Open games include all of these as special cases.

[3]Of course, these demands have been written retrospectively.