A summary of Out of the Tar Pit By: Ben Moseley and Peter Marks

Brandon Mainock

In simplistic terms you can subcategorize technology into two different fronts. Software and hardware. Over the course of the past three years a standard CPU went from a single threaded 4 core chip to 100% more lanes and cores per processing unit. Clock cycles went from 3.7 GHz max boost to 3.7GHz base clock, a top of the line clock speed 3 years ago is now in present day considered the norm to be relevant in the market. This was not just the case for CPUs but other hardware as well such as vector based lighting in Graphics cards and faster PCIE based storage devices. On the software side of things, over the past three years we have made marginal developments in machine learning, some would still define as being in it's baby stages, and video games that quite literally are defined as "bug infested". This is what is known as the "software crisis" where hardware-based solutions make marginal increases every year while software runs behind just trying to make sense and utilize the performance being given. This begs the question as to why software is always struggling to keep up with hardware developments and why is it failing to do so. The answer lies in the paper "Out of the tar pit" in which Moseley and Marks define software's hardest problem in advancing as simply complexity.

Complexity in a general sense is defined as a opposing force to comprehension by Moseley and Marks in which the root of complexity is based on a lack of understanding the system being used to solve a problem. They go on to talk about to find reasoning and comprehension we as a collective usually default to a trial and error testing strategy till we have enough data to make a general idea of what a system would do under parameters. This approach to understanding is flaw how ever in that state and the ambiguity that defines state makes testing a limited device. The alternative being Informal reasoning where a case is given in a simulation to enact a certain behavior from the system. However being limited to cognitive ability the more a system scales in size and numbers of state based variables the harder it is to define it as a viable option. State based variable especially offer a good portion of blame for complexity in systems as not only can they be ambiguous in design and use but they are not limited to use cases in scenarios that could be thought of in informal reasoning. This unlimited potential for state based variables makes them a powerful source for introducing complexity to systems, it is this reason that Moseley and Marks make the claim that we should do all that we have in our power to limit and isolate state in a system as possible.

The next biggest culprit of complexity is based in control logic in which a programmer is forced to use a language with ill defined flow. To create a process in which through manipulation a system does what is intended rather than being told what is wanted as a result. To be human is to error and using a system based on humans and their fault of ambiguity in either creating duplicate functionality or by not knowing what an explicit term does and then trying to use it as intended offers a door to complexity. This is especially true the more powerful a language is as user errors in a program can be stopped at a certain stack however a misuse in the kernel using interrupts and throwing exceptions wrong can only add more variations of misuse of explicit terms. This theme continues in object-oriented programming in which objects are state based as well as a fundamental building block to the functionality of systems based on it. It is because of this that limiting OOP in terms of state is an extreme undertaking in which there is no solution now. Moseley and Marks make it very clear that it does not matter how much complexity starts in a system since even a small amount of complexity will breed ever more complexity.

So what is the answer to these state using/based languages? Functional programming, a technique based on avoiding state and introducing mass referential transparency, the ability to replace functions with a value. This, in contrast, is preferable to an encapsulated state around making it a puzzle piece fitting into different functionalities and parameters which can become complex very easily compared to functional programming's approach of avoiding implicit/mutable state. This is all grand but the main flaw that functional programing must deal with is the fact that even with the intent of avoiding state, inside of a big system functional programming is built to maintain a certain state.

Now that we have established that complexity is everywhere and that even our solutional languages still suffer from side effects of complexity the best way that has been found to deal with complexity is to define and subcategorize the different types of complexity. The first being Essential complexity this exists as the user/programmer's problem needing to be solved and figuring out the system needed to solve said problem. The other being accidental complexity, to which any other amounts of complexity are categorized into. The major difference between the two types is that essential complexity will always exist even in an "ideal world" in which there is no accidental complexity. An example would be you are trying to make a video chatting service and as you keep hiring devs you get different amounts of comprehension from the people building the system. This can lead to duplicate functionality like writing two different methods meant to take a user to a home screen saved in different parts of the build. This complexity is avoidable put not inherent in the undertaking of a project in an ideal scenario. This subcategorizing complexity is not meant to fix the complexity issues but rather to define them.

State, along with complexity, also has an accidental definition in which any state used that could have been functional in an ideal world is considered accidental state. An example of accidental state is data being provided to the system since there is no way to tell what will be inputted or given to said system due to edge cases. Since languages in control logic suffer from ambiguity of state variables/functions in that there are no simple formal requirements Moseley and Marks consider all control logic to be accidental complexity.

Even though complexity seems to be the poison in the drinking water the fact of the matter is that even Moseley and Marks found functional causality for use of complexity. The two examples were performance sake and ease of expression. An example in terms of performance is multiprocessing in python in which you create a type of object/state where a process runs on another thread/core. This can cut times of computationally intense tasks by as much as 15% or more. As for ease of use having a name of a state such as helper_method_count can be a good way of making a program with state easier for a human to comprehend. The only caveat is that by introducing complexity it should be common practice to follow the steps of isolating and not using it unless necessary.

This leads finally to the optimal execution of a functional relational programming-based system or FRP system for short. In this system there are three basic parts the essential state, the essential logic and the accidental state and control. Essential state is defined as the foundation of the system being built, described as the purpose, essential logic is what is needed to create said result, and accidental control is how you use the essential logic to get the desired result. For comprehension lets assume you want to build a fire, your essential state is that you want to build a fire, your essential logic is knowing what you need to make a fire in terms of supplies such as wood, flint, and steel. While the accidental state and control is the actions that must happen for the supplies to create fire. This translates to CS in that the essential state is your defined purpose of system, not based on state, your essential logic is like a csv file of data but also has functions that can be used with data like a specialty use tool, and finally the accidental logic and control would be a declarative language using those tools in essential logic like SELECT operators for mySQL. Although FRP is still just hypothetical, advancements and attention increase as functional programming and its advantages gain more popularity with every year.