SAND2021-14541C

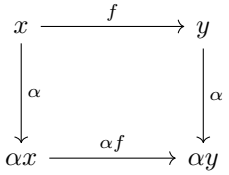
Toward a Science of Abstraction Design in Software

Kirk Landin (SNL) ktlandi@sandia.gov

Abstractions are the core building block of software. Good abstractions will allow software to quickly grow and evolve to meet the ever-changing needs of the research community. Bad abstractions will cause software to collapse under its own weight, as it becomes brittle, inflexible, and riddled with bugs. The overwhelming majority of literature on software quality focuses on improving the software engineering process and not the actual design of the software. Process is important in software engineering, but is only part of the picture. Poor engineering process can successfully be retooled at any point in a software lifecycle, superficial design issues such as non-uniform indentation can be fixed with simple text-processing tools, however software with a poor abstraction design is much harder to fix and may have to be re-built from the ground up to overcome its problems and limitations. Because of this, the structural design of the software and its abstractions is one of the **Most Important** facets of its construction, yet most of the software engineering community never mentions it as an issue. This needs to change!

What is an abstraction, Really?

What is an abstraction, really? Ask five different computational scientists and you will likely get five different answers, because abstraction has never been taught in a principled manner. An abstraction is a logical theory that accurately describes the structure of a computation to some level of precision. At its core, an abstraction is a *Commuting Diagram*, as shown here.



At the top of the figure is the *Concrete Domain*, with data values x, y and operation f. The bottom of the diagram contains the corresponding *Abstract Domain*, with data values, αx, αy, and operation αf. The *Abstraction Operation*, α, maps every concrete value/operation to some abstract one. It essentially "projects" away most of the unneeded details about the concrete domain, leaving a "distilled" interface that contains only the details necessary for the required abstract reasoning.

The abstraction operation, α, is *Sound* if this diagram commutes for all possible concrete values, x. This means that **All Possible Behavior** of the concrete implementation conforms to its abstract model. Soundness is an incredibly useful property when designing systems, especially ones that have many layers of abstraction.

The two foundational mathematical formalisms for specifying abstractions are *Functions* and *(Co)Algebraic Theories*. *Functions* (pure functions, in the mathematical sense) let us describe self-contained operations, and *Algebraic Theories* (along with their dual *CoAlgebraic Theories*) let us describe inter-dependent operations. When developing (Co)Algebraic Theories, one needs to specify behavioral invariants, which cannot be done in any commonly-used programming language. Because of this, developers should leverage modeling tools that enable them to specify/model/check invariants. *Alloy* is a very mature tool, developed at MIT, that allows very efficient specification and verification of abstractions and invariants. It is especially suited for structural modeling of software (Jackson 2012). *TLA+* is a suite of tools, also very mature, for modeling abstractions, and it is especially suited for modeling of state machines and state transition systems (Lamport 2002).

Abstraction Design as a Scientific Process

The general process of scientific inquiry is that one formulates a hypothesis and then runs experiments to test that hypothesis. If the experiments disprove the hypothesis, the hypothesis is modified to incorporate the additional knowledge gleaned from the experiments. This process is repeated until the hypothesis is sufficiently precise and general to adequately model the phenomenon of interest. This process is the same in Software Abstraction design. An initial abstraction is posed to model a component, "experiments" are run by developing both concrete implementations and client code for the abstraction. The hypothesis is then tested by asking the two questions, "Do the concrete implementations model the abstraction?" and "Does the abstraction provide the necessary information to the clients?" This process continues as new concrete implementations and clients of the abstraction

are created. At any point in this process, one of the new components may disprove the hypothesis/break the abstraction. In this case a new, more correct, hypothesis/abstraction will be developed. At some point, the hypothesis/abstraction will become sufficiently useful and the hypothesis testing loop will stop.

In addition to experimental data, abstraction designers leverage their theoretical understanding of software semantics to make informed decisions about what hypotheses/abstractions they should propose. They use this knowledge to craft elegant abstractions that have desirable properties (compositionality, soundness, simplicity, generality, etc.) Just as we expect a Physicist to adequately understand the underlying theory and propose hypotheses that are adequately informed by this understanding, we should expect our Computational Scientists to do the same thing. However, very few Computational Scientists have any knowledge about Programming Language Theory, Type Theory, Formal Semantics, etc. Because of this lack of knowledge, many design decisions are ad-hoc, can be very sub-optimal, and may be highly damaging to the overall software system.

Research Challenges

Leveraging the Full Power of Functional Abstractions: *Functional Programming* has been a buzzword for at least a decade now, and for good reason too, as functional abstractions are some of the simplest, most composeable, and easiest to verify abstractions in software. Our current software stacks woefully under-use functional abstractions and suffer a lot because of it. How should we re-architect scientific computations and libraries to maximize the use of functional abstraction? How do we decompose our computations in a way that is amenable to functional abstractions, and what should the functional abstractions look like for these computations? Which of our computations' correctness properties work well to encode in a language's type-system, are verifiable by standard type-inference algorithms, and make for a pleasant user experience? How do we effectively educate scientific software developers on the structure and use of these abstractions?

Designing Good Non-Functional Abstractions: In most code-bases there will be a non-trivial amount of functionality whose structure does not fit within the confines of functional abstractions. For these portions of common scientific computations, what appropriate (Co)Algebraic Theories describe the behavior at the correct level of precision, are elegant, easy to reason about, and are composeable with other abstractions? What sort of tooling and workflow will help our developers be efficient and productive in developing and testing non-functional abstractions?

Automatic Verification via Property-Based Testing: For correctness properties other than those which can be encoded into the type-system, we need an additional step to verify that concrete components actually model their abstractions. One of the most effective tools for doing this is *Randomized Property-Based Testing*, in which invariants are reified as unit tests that randomly generate large sets of inputs, testing that the given invariant holds for each input. If none of the test cases disprove the invariant, then there is good (although imperfect) assurance that the concrete implementation actually models its abstraction. How do we define our invariants so that they can generate effective property-based tests? How do we automate this test generation process as much as possible? How do we randomly generate test data for our different components in a way that ensures proper domain coverage by these randomized tests?

Concrete Steps to Move Forward

We need to start treating abstraction design as a real science, and not as some lesser, throwaway, activity as it has been treated in the past. We need to start using the scientific method to build abstractions. Everyone in development needs to understand the basic mathematical formalisms of abstraction and understand how syntactic mechanisms in mainstream languages, such as classes, inheritance, and generics fit into these formalisms. We need a culture that encourages developers to be precise about their abstractions when needed, actually specify invariants, and use modeling tools such as Alloy or TLA+. We need the Programming Languages community to start teaching the Scientific Computing community about the mathematical foundations of abstraction design. One of the most effective tools for teaching semantic reasoning to programmers is for them to study a programming language whose structure is closely tied to the semantic foundations of software. Haskell is the best language for this, because it forces programmers to wrestle with the semantic structure of their code much more than other languages. This causes a programmer to develop an intuitive understanding of semantic structure that can be applied to software systems written in any language. A year of working in Haskell yields huge benefits for a programmer's ability to design things in Python and C++.

References

Jackson, Daniel. 2012. *Software Abstractions: Logic, Language, and Analysis*. MIT press.

Lamport, Leslie. 2002. *Specifying Systems*. Vol. 388. Addison-Wesley Boston.