Applied Compositional Thinking for Engineers (ACT4E)

Session 7 - Life is hard (but category theory is harder)

Questions & Answers

Q: consider computation, the flops/Watts depend on the implementation of the algorithm also, it may not be monotone?
AR: Why do you say flops/W depends on the algorithm? I would think it's just a property of the chip.
MH: chips have lots of different functions e.g. if you can use the GPU or if you use an accelerator etc
AR: I see. I don't think of the GPU as part of the same chip. But basically you're saying that not all floating point operations are equivalent, right?
MH: I'm saying it depends on how you implement the algorithm

GZ: Without going into the details (we will see that the theory of co-design has a notion of "implementation" which captures what you asked), here is what monotonicity says.
If given some flops F some power P is required, the very same power will also allow for less flops. Conversely, if you allow some more power, you will still at least have the implementation which provides F flops.

MH: OK, but once you start making functional trade-offs I'm not sure this is monotonic.
AR: I'm actually convinced now that it is: for any given implementation, being able to supply more power *will never* hurt.

GZ: answer to MH above. It is indeed monotonic, and we will see that the notion of motonicity is applied to the set containing different implementations. Let me explain better (but keep in mind that this will be covered in next lectures):

- Say that a set I of implementations satisfies the query of willing to produce F flops while requiring P power. If you reduce your expectations and choose F'<=F, the new set I' of implementations making you happy is S' supseteq S (at least the same)
- Conversely, if you decide to allow more power, i.e. P'>=P, you will again get a S''supseteq S.

You will see that the map which describes the above feasibility relations is a monotone one

MH: that feels right if you are selecting only based on F and power. But it seems that once different implementations have different functionality then one might need more F but that requires less power because the algorithm uses the chip in a more efficient manner.

GZ: you will see that design problems can have also multiple functionalities/resources. Really try to think about it in terms of set of ways to obtain F given R (set which increases in size if we restrict our expectations or if we have more resources)

AR: MH, I think you're thinking in terms of a holistic property; do you accept that, locally, it's monotone? That is, all else being equal, there's no situation in which having more power available makes a flops that was previously achievable no longer so?

MH: I assume the relation can be read in both directions i.e. a given F requires at least a certain power. There are more than those two variables in the example - functionality. If you ignore functionality or fix it then I agree.

AR: I would read the equation as follows: for any implementation which delivers F flops while consuming W watts, the same implementation can also deliver F' <= F flops while consuming W watts or consume W' >= W watts while delivering F flops.

MH: Sure, but the question was about different implementations - if we hold that constant it seems obviously monotonic.

GZ: to answer MH, it is not a single implementation which is frozen. Take a bunch of different implementations. Fix them and reason as above. If you add a new implementation, I don't see what you are trying to compare.

MH: It was the graph showing functionality that motivated the question. I was thinking that depending on that trade-off some implementations may consume less power for more flops. I'm happy to remove all of this if you like - as you say it might become clear later - I have basically no idea what co-design does.

AR: Obvious properties are the best properties! Monotonicity like this does *not* always hold for different kinds of systems (particularly ones with feedback loops) If you have multiple different implementation options (a choice, if you will), each of which is monotone, then the choice preserves monotonicity. This is only a property about two variables, though.

Q: Does the LUB and GLB of a *finite* set in a total order always exist?
GZ: Yes

Q: Are functors necessarily functional relations, i.e. not one-to-many?

AR: I believe so. I'm sure there's a generalization that allows multiple outputs, though.
JL: Yes, functors are not one-to-many. They are based on functions. We will see that profunctors later in the lecture. A helpful analogy is: profunctors are to functors as relations are to functions.

Q: Do functors have similar properties as functions, i.e. injective and surjective, how are they called?
AR: Faithful and full, respectively, IIRC. (NM: not exactly)
NM: in a category monomorphisms and epimorphisms are the generalisation of injective and surjective, since you have a category of categories where your morphisms are all functors, then the natural generalisation of injective and surjective are functors that are monomorphisms and epimorphisms. Functor from C to D can instead be Full or faithful which means they are surjective or injective on each Hom-set from X to Y in category C (This is not necessarily on objects and fully faithful functors may therefore merge different Hom-sets together meaning there may not be an inverse mapping on the morphisms) https://en.wikipedia.org/wiki/Full_and_faithful_functors

Q: What would be an example of no functor where the composition law is not satisfied, ie: F(g);F(f) /= F(g;f)?
A functor always has the composition law, so if that is broken it is not a functor. However there are two functors where the composition law is now the existence of the 2-morphism in the target category $\gamma_{f,g}: F(g) ;F(f) => F(g;f)$ for every composable pair f and g from the source category.

AR: If you know Haskell, Haskell Sets are not functors. More generally, this occurs when you "only sort of forget" some structure
NM: However I have heard that Haskell is not a category
AR: To the extent to which this is true, it is not relevant for this example.
NM: Thought I raise this point as it may be confusing to get different answers from different sources (a mathematics source would say you need composition), furthermore if you don't assume that composition has to hold you may have a theorem that doesn't hold in your example.
AR: I *think* that translating the Haskell example to math, you get "a Setoid with over an equivlence relation that is coarser than the ambient equality relation"; for example, a function which maps numbers to the smallest even number <= than them, lifted to work on collections.
NM: My background isn't maths so this doesn't help, my statement is repeated from a lecture I had on category theory. If you are saying that there exists something named a functor in haskell that doesn't behave like a functor then that may be the reason for the statement that Haskel isn't a category.
AR: I see. Yes. For the purpose of this example Haskell "Functors" should be thought of as loosely analogous to actual functors; however, the failure of composition illustrated in the code applies whether we're talking about "Functors" or functors.
NM: by definition it is not a functor from category theory then.
AR: The intent was to use Haskell syntax to convey the math concept. I find it more illuminating than if I had to translate the example into pure math, where it still applies.
NM: The key thing I wanted to make sure was conveyed is that there *does not exist* a functor F where **F(g);F(f) /= F(g;f).** You can find examples of functor like things but while it sounds pedantic if you call something with this property a functor then theorems you can pull out of a book may not hold

AR: Of course not! That's not a functor! By definition! :) The Haskell example is pointing out what goes wrong when you try to claim that a specific functor-like-thing is a functor, providing a counterexample to the composition requirement, proving that although you can "make it an instance of the Haskell typeclass Functor", it's not a functor. This matters for Haskell programmers because it's a violation of the implicit contract of the typeclass, but that's an aside.
Let me try to turn the Haskell into math and see what I come up with.

Q: (leading pedagogical question) If we have a poset P with no meet and join, how do we create from P a new poset that is a lattice Q (ie where the meet and join now exist in the poset)?
KL: So on slide 16 Upper/Lower closure, maybe we should mention that upper closure maps posets to lattices.

Q: Can you define all/some categories out of an empty category, similarly how natural numbers can be defined as nested sets starting from an empty set?

JL: As Andrea explained, we can "model" the natural numbers using categories. And many properties familiar form the natural numbers carry over. I'm not sure if perhaps there was also another aspect to this question, though. One other thing to say is that there are various typical constructions that, given some categories, allow you to build new ones. Analogous to sets: with sets we can form products, sums, powersets... etc. and also we can, given sets X and Y, consider the *set* of all functions from X to Y. Similar things work for categories. However, the world of categories is *not* like the natural numbers in that one could build *all* categories starting from some primitive category.

I believe what you are looking for is in part covered by the idea of periodic table of categories (https://ncatlab.org/nlab/show/periodic+table) where you can start with truth values and build these to sets, then sets can be used to build all 1-categories, then 1-categories can be used to build all 2-categories and so on. Then you can place extra restrictions on some of the morphisms (or n-morphisms) to obtain other types of categories (e.g. monoidal categories). I will note that I believe there may be categories outside this, currently there are multiple ways people can extend 1-categories to higher categories and I believe the questions of if these are the same or different or valid are still open problems for category theory.