Structural Analysis for Logicians

by Sven Nilsen, 2024

In this paper I introduce a basic framework for logicians that circumvents the usual bottlenecks of combinatorial explosion in structural analysis and the need for generics to make results applicable.

Mathematicians have long time struggled with defining a proper formal structure to reason about structures in general. Apparently, all such designs have to come with some tradeoffs. For example, in programming language theory one has the problem of implementing algorithms correctly, which is not the same problem as getting correct proofs for abstract theorems. The solution so far has to split mathematics roughly into two distinct fields: Applied and Pure math.

However, Path Semanticists want to prove theorems for structural analysis that might inform decisions on mathematical language design. This is a very different need than either Applied or Pure math. The approach taken in Path Semantics is often to test out ideas in different mini-languages and iterate successful designs. This is why logic is frequently used, because it allows a such approach without needing a large and complex framework.

This means there is a need for a basic framework for structural analysis in logic.

In most of mathematical history, people have been biased toward their own favorite structures. It is easy to make this mistake, because people are naturally inclined to trust something they are already familiar with. Depending on how people learn math, they develop different tastes for design. However, the problem is that what might fit one application for one person might not fit another application for a different person. Researchers are also more reluctant to share the downsides of their own designs, something that makes it more difficult for new people to decide which design to go with. This is why is important to get at the core of structural analysis in logic and not just selecting an arbitrary approach.

Structural analysis is a complementary discipline to other approaches in mathematics. For people who are not accustomed to this way of thinking, it can appear to be very strange at first. It takes some time to get used to the ideas, however, when it finally "clicks" one can start applying this approach to many sorts of problems.

In programming, when a structure is defined, the entire purpose is that the structure is known to the compiler which verifies that the programmer uses it correctly throughout the codebase. However, in structural analysis, the point is not to know in detail how the structure is implemented. This makes it difficult for programmers to initially see why structural analysis might be useful. To explain how, I will start with the \`enum\` type in Rust and show how a simple modification turns it from something that is useful in a typical program to something that reflects structural analysis.

```
pub enum Expr {
     Prop(String),
     Imply(Box<Expr>, Box<Expr>),
     And(Box<Expr>, Box<Expr>),
     Or(Box<Expr>, Box<Expr>),
}
```

In this structure, one can express propositions like \`a, b, c\` and logical binary operators \`a => b, a & b, a | b\`. One can also do nested expressions like \`a => (b => c)\`.

Now, I will modify this structure such that all “content” is moved to the type level:

```
pub enum Expr<A, B> {
    Imply(A, B),
    And(A, B),
    Or(A, B),
}
```

Notice that this changes the entire “feel” of the structure, but it still allows the same amount of expressiveness. In fact, it allows more! The structure is no longer restricted to using `String` for the name of a proposition. However, each time you use the structure, you will get a different type, which is not good for the size of the binary produced by the compiler. The downsides to this design are plenty, so most practical programs utilizes a similar design to the first example instead.

A mathematician might also object to the above example because it is not closed over some set of objects. For example, in a Category, there is a set of objects and a set of morphisms. It feels natural for a mathematician to work with a closed universe of things which are of concern to some problem. When the structure is opened up like above, it feels a bit incomplete.

Yet, there is a reason why it is interesting to do structural analysis this way. In structural analysis, one is not particulary concerned about what the structure does or how it is implemented, but more about how structures relate to each other and how efficient they are at representing a solution. When we have a complex mathematical object, from a perspective of structural analysis we are not that interested in the details of the object, but whether it fits inside the expressiveness of the structure. One structure can be used to represent many different objects, but when the design is changed some new objects might be expressed, while some objects might no longer be possible to express or more difficult to express than before. Structural analysis allows us to compare different designs and even find new designs using automated theorem proving techniques.

It is not easy to use a general purpose programming language to do structural analysis with these kind of open structures, because we would have to code all the properties that are needed for doing the analysis. The point with using a framework is to avoid needing to recreate the tools each time.

The basic property of any structure is that it is an object with something inside it that might be used for different purposes. If the structure has no room for what we need, then we have to come up with another structure design or another approach. We might also be interested in how much effort it takes to make some structure useful. Structural analysis deals with these kinds of situations by providing answers to questions about structures.

If a structure does not have any ability to represent something inside it, then it is not useful and therefore in general not considered a structure at all. This means that it is usually given that a structure can represent something, although we do not always specify what it is. At first sight, this seems to be a problem on its own, because how can we reason about something if we do not know what it is?

The trick is to realize that in mathematics, most of the things that we use math for is generalized over specific details about objects. For example, when using equality, we are often more concerned about the algebraic properties of equality (reflexivity + symmetry + transitivity) than about what kind of objects the equality compares. So, most of the time, we do not need to know much about the objects that are the end points of the more abstract algebraic properties or relations between them.

If there is no need to know something, then one can exploit this property in mathematics.

In summary so far, one does not need to know:

- What set of objects or data that one operates on
- How to close the structure over some universe of objects
- Which programming language or how the structure is implemented

In structural analysis, one takes these 3 properties and “squeeze out” every drop of performance.

As a thumb rule, if there is no need to know in detail what the structure is used for, then one can assume that the structure is a binary tree, because any data structure can be represented as such.

Another thumb rule is that one does not care specifically about what the nodes in the binary tree does. However, there are other properties one cares about, such as the complexity of the binary tree in order to represent some object.

It is also important that one can use algorithms to traverse the structure. Otherwise, the structure is not very useful. So, there are some objects that one puts inside the structure, for example two objects \`a\` and \`b\` which represents the left and right child node respectively and one puts a structure on them that is later used for reasoning. Now, you might think of something like a list \`[a, b]\` which is understandable, because you would think that if you have \`a\` and \`b\`, then you can put them together. However, \`a\` and \`b\` in structural analysis functions more like references. You do not get to “have” \`a\` and \`b\` explicitly. The structure is more like source code where things like lists are just one possibility. The convention of using \`a\` and \`b\` is that they are two references which might refer to the same underlying object. By understanding them as references, one can think of getting access to \`b\` depending on the access to \`a\`, e.g. \`a => b\`. It takes some time to get used to this idea, because in the physical world we are used to pointing to specific objects. A structure can be more abstract, more like telling you how to navigate a landscape than how to organize objects.

What makes structural analysis so powerful is precisely the ability to forget or not care specifically about what it means in simplistic terms. However, when reasoning about structures, one usually specifies how the structure works on a logical level. We do not care about sets or categories as objects, but we care about how we can start with some simpler structure and create sets or categories out of them. The logical properties of structures is very important.

In order to make structures composable, one builds up structures using logical operators. In constructive logic, there are only 3 basic binary operators \`=>, &, |\`. You can add more operators such as \`^, :, ~~\` later, but this is not important for now.

- The \`=>\` operator is a map, like a lambda or closure
- The \`&\` operator is like a set of two objects
- The \`|\` operator is like a choice between two objects

Depending on the application, the specific implementation of such operators can vary. For example, there can be side effects that makes the structure useful. This can be a problem since logic does not have side effects. Yet, it is sometimes possible to ignore side effects when reasoning about the structure. Just because an application uses a structure in some particular way does not usually prevent us from performing structural analysis.

The usefulness of logical operators is that one can use theorem provers to solve problems. Logical operators also provide a common language that lets us compare designs of structures. Another benefit is that by doing analysis closer to the language of logic, one often improves the design. Many software engineers over-engineer their data structures because they do not know enough.

Now I will put these things together to define a structure formally.

A structure is a triplet `(P, a, b)` such that the first component `P` is the name of the structure, followed by the second component `a` which is the first object that the structure operates on, followed by the third component `b` which is the second object that the structure operates on.

In structural analysis, it is common to write `P(a, b)` for the structure `P` on `a` and `b`.

The 3 basic binary operators in constructive logic can be used as structures:

- P(a, b) == (a => b)
- P(a, b) == (a & b)
- P(a, b) == (a | b)

It takes some time to get used to the idea of not caring about the specific structure. However, it is also important to keep in mind that when we choose a structure, this will give the structure different properties depending on our choice. It is not like the logical properties "vanishes" or something!

Now, as an exercise, try expand the following using the 3 choices:

    P(a, P(b, c))

One gets the following:

- (a => (b => c))
- (a & (b & c))
- (a | (b | c))

If you have 3 objects `a, b, c` and you want a structure where you can have one of them at a time, one can use `P(a, P(b, c)) == (a | (b | c))`. So, this naturally wraps each of these 3 objects into the structure where they become part of a larger whole, but still they are functioning as individual objects and not as a collection. The trick is that one can say `(a | (b | c))` and you do not know specifically which object is used. However, you can try each one in turn and see what happens.

The `|` operator is the most important one in structural analysis, because it lets people create a kind of "grammar" for the structure they are using. The second most important operator is `&` which functions kind of like a set. However, there is a sense in which both of these operators behave like sets, with the difference that they provide different perspectives of the same underlying set.

Putting `|` and `&` together:

    P(a, b) == ((a | b) | (a & b))

Now, as an exercise, try expand the following:

    P(a, P(b, c))

The solutions are `(a | (b | c)), (a | (b & c)), (a & (b | c)), (a & (b & c))`.

Notice that if you have objects `a, b, c` then you can put each of them into `P(a, P(b, c))`.
This is strictly more expressive structure than `P(a, b)`. When you are not used to structural analysis, it is difficult to understand that one can expand many structures this way.

In one sense, we took out the content of the structures as we use them when implementing a normal computer program. However, we can keep the structure that makes analysis possible!

Another way to think about this, is when one blows a balloon. The balloon, when inflating, must occupy some space. If there is no room for the ballon to grow in size, then it does not matter how hard we try to blow it! So, we make room for the balloon to grow. The structure is kind of like the room where the balloon grows. We do not care in particular about the size of the balloon, only whether it fits inside the room or not.

One important property of a structure is whether it is commutative:

P(a, b) == P(b, a)

Commutative structures are easy to work with when extending them, using themselves.

Another important property is the following, which is called “right extensible”:

P(a, b) => P(P(a, b), c)

Similarly, left extensible is defined as following:

P(b, c) => P(a, P(b, c))

A third important property is associativity:

P(a, P(b, c)) == P(P(a, b), c)

A permutative structure can swap the arguments in any order.

When a structure commutative and associative, it follows that it is also permutative. If it is left extensible then it is right extensible and vice versa, so in that case we call them just “extensible”. These structures are much simpler to work with than others, because you can just extend it in any way and it does not matter how you do it. This means, one can treat it as a structure on some set and any extension to the set is easily managed.

If it does not matter to the analysis on a structure whether it is slightly larger, then it is often desirable to make it commutative and associative, when possible.

For example, the following structure:

P(a, b) == (a => b)

This is not commutative and associative, so by extending it to the following:

P(a, b) == ((a => b) | (b => a))

This saves us a lot of headache later when doing structural analysis.

In general, one can often improve a structure in the following way:

P’(a, b) == (P(a, b) | P(b, a))

As an exercise, one can prove the following structure is commutative:

∵ P(a, b) == ((a => b) | (b => a))
∴ P(b, a) == ((b => a) | (a => b))
∴ P(b, a) == ((a => b) | (b => a)) reorder
∴ P(a, b) == P(b, a)

Prove associativity is much harder, so I checked it with the Pocket-Prover library (classical logic). However, I would like to have a constructive proof too, which is left as an exercise to the reader. Perhaps you can do it with Prop or Hooo?

When we extend the design of a structure, one can often treat it as following:

P’(a, b) == (P(a, b) | X(a, b))

By default, `P(a, b) => P’(a, b)`.
However, there is some `X(a, b)` such that if `!X(a, b)`, then `P’(a, b) => P(a, b)`.

With other words, we can put the old structure into the new one and we can retrieve the old structure from the new one if it is encoded into the new one.

This tells us that `P(a, b)` is “in” `P’(a, b)`.

Now, I will introduce a new ternary operator for “in”:

in(P(a, b), P’(a, b), X(a, b)) == ((P(a, b) => P’(a, b)) & (!X(a, b) => (P’(a, b) => P(a, b)))

A common trick in structural analysis is to remove the arguments of structures when they all operate on the same arguments and lower them to propositions. This improves solver performance.

in(P, P’, X) == ((P => P’) & (!X => (P’ => P))

The “in” operator is a natural ternary operator for structural analysis.

One can prove `in(P, P, true)` and `in(P, P, false)`.

Conclusion

The benefit of using a basic logical framework for structural analysis is that one avoids the usual bottlenecks of combinatorial explosion. It is also not necessary to use generics in some programming language, as logic naturally leans toward generics.

It might seem counter intuitive that what a structure contains ought to be parameterized when doing structural analysis, but this is something that vastly improves performance of solvers.

There are some common tricks that one can use to simplify structural analysis. Ideally, a structure should be commutative and associative, such that if it is extensible then any order of extending the arguments is equivalent to all the others. Such structures can be thought of as operating on some underlying set of objects.

The “in” ternary operator is natural for structural analysis.