Reasoning About Molecules

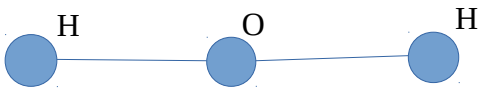
by Sven Nilsen, 2017

This is a practice paper to build an intuition around the mathematics of molecules, such that it can be used in automated engineering.

In chemistry, a common way of representing a water molecule is:

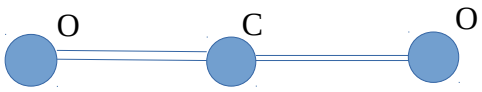
H_2O

Another way is to make a drawing where angles are ignored:



The water molecule is naturally structured this way, because the molecule is held together by electrons forming pairs in the outermost shell. The atom number of `O` is 8, which means it has 6 electrons in the outer shell, since there are 2 in the innermost shell. The atom number of `H` is 1, which means it has 1 electron in a single shell. By each `H` atom sharing an electron with the `O` atom, while getting a share one of its electrons in return, the atoms form the chemical stable structure we call a “molecule”.

Here is a drawing of the molecular structure of CO_2:



The atom number of `C` is 6, which means it has 4 electrons in the outer shell. Notice that instead of sharing a pair of electrons, the `C` atom shares two pairs with each `O` atom. In total there are 4 pairs of electrons that make up the chemical bound of the molecule.

One can formula the following general problem: How do you determine whether any such graphical representation can represent a physical molecule? We would like to have an algorithm that can tell us what real molecules look like.

A key insight is that when using objects to represent atoms, the properties of these objects are very complicated, and so will the algorithms be that reason about the molecules this way. Instead, one should think of the whole molecule as a structure consisting of two things:

1. A symmetric matrix of size NxN representing the number of electron pairs between N atoms
2. A vector of size N assigning an atom number to each atom

The order of elements in the vector is kept such that we can look up in the matrix to see which atoms are connected. The symmetric matrix can be generalized with vectors to store chirality and shapes of molecules, but the rest of this paper I will use scalars.

A matrix is symmetric when the following function returns true (as defined in the Standard Dictionary of Path Semantics, link https://github.com/advancedresearch/path_semantics/issues/129):

```
sym := \(m : matrix ∧ [dim] [eq] true) = ∀ i, j { m[i][j] == m[j][i] }
```

In path semantics, one can write sub-type constraints that are defined using functions. This makes path semantics a very powerful tool for reasoning, because a lot of mathematical ideas can be expressed directly. The `[dim] [eq] true` constraint means the dimensions of the matrix `dim : matrix → (nat, nat)` is checked for equality `eq : A × A → bool`. When this check returns `true`, the matrix are of dimensions NxN, called a "square matrix". One could also write it as a composition `[eq · dim] true`, or define a new function logically equivalent to this one.

It follows that all symmetric matrices are square matrices, but are all square matrices symmetric?

If we think about the `sym` function, we know that it returns `true` for some input and `false` for some other input. Therefore, there are some matrices that are square matrices but not symmetric. In path semantics, one expresses this knowledge as an "existential path" of `sym`, a function that tells what output is returned.

```
∃sym <=> \(x' : bool) = if x { true } else { true }
```

This is a function called `$true_1$` in path semantics, because it ignores the single argument input and always return `true`. It is written `$true_1$` for brevity.

```
∃sym <=> true₁
```

Existential paths are important because, without them, we do not know whether a constraint makes sense at all. They are the type checkers of path semantics. The brilliant thing about path semantics is that it grounds meaning in functions, such that it is a testable and sound system for reasoning without being constrained by the conceptual difficulties we are dealing with, which can get quite complex because we are reasoning about molecules.

When we write the following:

```
a : [sym] true
```

It means `a` has a sub-type defined by the `sym` function. However, you should not get yourself tricked by the simple and expressive notation. In path semantics there are lots of machinery behind the scenes.

We know that in order for it to make sense to talk about a symmetric matrix, it must be a square matrix. To make this a meaningful statement we need to check two things:

```
a : [dim] [eq] true
a : [sym] true
```

The first sentence can be written as a composition:

```
a : [eq · dim] true
```

This is meaningful if and only if:

    true : [∃(eq · dim)] true

The existential path must return `true` for the output `true`. Equality returns `true` for some input and `false` for some other input, so the existential path is logically equivalent to `$true_1$`:

    ∃(eq · dim) <=> $true_1$

Replacing `∃(eq · dim)` with `$true_1$` gives us:

    true : [$true_1$] true

This process can continue forever, because:

    ∃$true_1$ <=> id
    ∃id <=> $true_1$

However, since we can use this as hardcoded knowledge, we assume that it will continue to pass the checks forever, so we determine that `a : [dim] [eq] true` is meaningful.

The next to check is:

    a : [sym] true

Which is easy:

    true : [∃sym] true
    true : [$true_1$] true

An existential path lets us determine whether it is meaningful to talk about variables having a sub-type. Path semantics divides up the things to check into neat sub-problems, and when all these sub-problems are solved we can use the expressed knowledge to reason about more advanced stuff.

A symmetric property is not the only constraint a matrix must have to represent shared electron pairs in a real molecule. However, it demonstrates the conceptual idea that no matter what structure, form or shape a molecule has, it can be represented and reasoned about using path semantics.

One property that some atoms have is that they want 8 electrons in their outer shell and are not as willing to share the electrons in the inner shells. Thus, there could be a limited number of shared electrons between any two atoms in a molecule at a given pressure and temperature. When the molecular structure is limited by shared electron pairs, one can use the following constraint:

    max_bounds := \(n : nat) = \(m : matrix) = ∀ i { ∑ j { m[i][j] } <= n }

This is used this way, by passing an argument to the constraint:

    a : [max_bounds(3)] true

Another constraint is that you might want to describe structures where every atom is connected by sharing at least one electron with another atom. An exception to this could be a complex molecule that transports a single atom by containing it. When every atom is connected to something else, the following function can be used:

each_connected := \(m : matrix) = ∀ i { ∑ j { m[i][j] } > 0 }

An atom can not be connected to itself:

non_diag := \(m : matrix ∧ [dim] [eq] true) = ∀ i { m[i][i] == 0 }

Like with the other constraints, you can define a sub-type like this:

a : [non_diag] true

By combining the constraints, you can define a more precise sub-type:

a : [sym] true ∧ [max_bounds(3)] true ∧ [each_connected] true ∧ [non_diag] true

To make sure this is meaningful, every constraint must be checked against the constrained existential paths:

true : [∃sym{[max_bounds(3)] true ∧ [each_connected] true ∧ [non_diag] true}] true
true : [∃max_bounds(3){[sym] true ∧ [each_connected] true ∧ [non_diag] true}] true
true : [∃each_connected{[sym] true ∧ [max_bounds(3)] true ∧ [non_diag] true}] true
true : [∃non_diag{[sym] true ∧ [max_bounds(3)] true ∧ [each_connected] true}] true

All the constraints are overlapping, so there is at least one matrix having all these properties. Another way to simplify the constraints is to replace them with a single function `f`:

f := \(x : matrix) = sym(x) ∧ max_bounds(3)(x) ∧ each_connected(x) ∧ non_diag(x)

a : [f] true

The goal is to get as close to a hypothetical function `molecule` that determines real molecules:

f <=> molecule

However, this might not be easy, so one can attempt another thing instead. By splitting the function into two parts, one that tells whether a structure is a molecule when it really is a molecule, and another that tells whether a structure is not a molecule when it really is not a molecule:

molecule(_ : [f_{true}] true) = true
molecule(_ : [f_{false}] false) = false

Try to eliminate all cases where our knowledge is uncertain:

a : [f_{true}] false ∧ [f_{false}] true

While avoiding unsoundness when these predictor functions are contradicting each other:

$$a : [f_{true}]\ true \wedge [f_{false}]\ false$$

The function `$f_{true}$` can e.g. be a list of all known molecules. The function `$f_{false}$` could be something like the one developed above, where the molecule structure is constrained by certain properties.

Once you have two such functions, one function that predicts positive cases and another function that predicts negative cases, you can use a constraint solver to find solutions for either unexplored molecules or any unsoundness in the assumptions.