A Scala-based Domain Specific Language for Stoichiometry

Course project for *Engineering of Domain Specific Languages*
Università della Svizzera italiana

Valentin Aebi

Autumn semester 2025-2026

1 Introduction

Stoichiometry, namely the computation of the quantities involved in a chemical reaction, is fundamental to the quantification of chemical processes. It is also one of the core topics of basic chemistry classes, such as the ones taught in high schools. Recurring tasks in stoichiometry include computing the mass of molecules, converting between distinct units of measure, and balancing chemical equations. These tasks can be tedious and repetitive, and often boil down to reasonably simple mathematical problems, which motivates the development of techniques and tools to automate them.

This report describes a prototype of such a tool, under the form of a domain-specific language (DSL) built on top of the Scala general purpose programming language. The purpose of this DSL is to serve as a formal but intuitive way of describing chemical problems such as *What is the amount of carbon dioxyde produced by the combustion of 200g of glucose assuming that the efficiency of the reaction is 60%?*. It also includes a resolution engine that performs the necessary computations to solve such problems. The focus of this report is on the language and its integration inside Scala. It also includes a brief explanation of the chemistry concepts that are necessary to understand how the language can be used, as well as a high-level description of the algorithmic aspects of the resolution engine.

The GitHub repository containing the implementation of the DSL, together with a few examples, can be found at `https://github.com/ValentinAebi/scala-chemistry-dsl`.

2 Background and algorithmic notions

2.1 Domain-specific languages

A domain-specific language (DSL) is a computer language designed for a specialized usage related to a specific domain, like stoichiometry in the present case. Its purpose is to make coding-based problem solving more easily accessible to domain specialists, whose knowledge of programming or time to invest in coding may be limited. DSLs do so by bridging the gap between general-purpose programming languages, which offer great expressiveness but are non-trivial to learn, and traditional software, which is often not flexible enough to support use cases that the software has not been explicitly designed for.

2.2 Basic concepts of stoichiometry

This section describes the necessary chemical concepts starting from the motivating question given in the introduction:

What is the amount of carbon dioxyde produced by the combustion of 200g of glucose assuming that the efficiency of the reaction is 60%?

The chemical reaction mentioned here, the combustion of glucose, is given by the following **equation**: $C_6H_{12}O_6 + O_2 \rightarrow H_2O + CO_2$. It describes how the **reactants** (glucose $C_6H_{12}O_6$ and dioxygen O_2) are transformed into **products** (water H_2O and carbon dioxyde CO_2). The amount of a reactant or product can be given either as a mass, for instance in grams like here, or in *moles*, a chemical unit corresponding to a fixed (very large) number of molecules or atoms and meant to avoid directly representing quantities in terms of the number of molecules, which would be impractically large. Converting between mass and moles and vice-versa implies computing the **molecular mass** of the molecule, which is the mass (usually in grams) of one mole of the said molecule. This mass is computed from the average mass of atoms, usually looked up in a table.

Before performing any calculation using such an equation, one has to **balance** it, that is, compute **stoichiometric coefficients** such that the number of atoms of every chemical element is the same on both sides of the equation, since the reaction has to obey the law of conservation of mass. In the example above, the balanced equation is as follows: $C_6H_{12}O_6 + 6O_2 \rightarrow 6H_2O + 6CO_2$ (the absence of a coefficient for $C_6H_{12}O_6$ means that the coefficient is 1). Some molecules may also have a charge (e.g. Na^+), in which case an equation is only balanced if the weighed sum of the charges is identical in both members of the equation. Finally, the **efficiency** of the reaction is a mean of quantifying the difference between the theoretical and practical amounts of products.

2.3 An algorithm for balancing chemical equations

While simple chemical equations can often be balanced manually using a trial-and-error process, it is also possible to reduce the balancing problem to a system of equations, and to solve it using linear algebra methods. Given a chemical equation, one can balance it by first mapping it to a matrix where each row corresponds to one of the chemical elements involved in the reaction, and each column to one of the reactants or products. Note that the quantities of reactants and products must have opposite signs. One can then use Gauss pivot-elimination algorithm to turn the matrix into a reduced form, from which the coefficients can be easily derived. Figure 1 shows this process for our example reaction.

3 Design and implementation of the domain-specific language

3.1 Design overview

Our chemistry DSL is (mostly) an *internal DSL* in that it is designed on top of Scala. This solution has the advantage of allowing the DSL to interact with the -more powerful- host language

$$\begin{bmatrix} & (C6H12O6) & (O2) & (H2O) & (CO2) \\ (H) & 12 & 0 & -2 & 0 \\ (C) & 6 & 0 & 0 & -1 \\ (O) & 6 & 2 & -1 & -2 \end{bmatrix}$$

$$\xrightarrow[\text{Gauss's algorithm}]{} \begin{bmatrix} (C6H12O6) & (O2) & (H2O) & (CO2) \\ 6 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

Figure 1: Matrix corresponding to the example reaction, before and after reduction. The reduced matrix essentially shows the relationships between the coefficients in a simpler way than the original matrix. For instance, it is clear from the first line that the coefficient of $C_6H_{12}O_6$ has to be $\frac{1}{6}$th of the coefficient of CO_2. It then only remains to perform a bit of arithmetic to find the minimal coefficients such that all of them are integers.

when needed, by exposing a programmatic representation of the objects it manipulates. The main drawback is a more complex design that must take into account the syntax constraints of the host language, which may impede the expressiveness of the DSL. The only exception to the internal nature of the DSL is the representation of molecules: since I was unable to find an intuitive way of representing molecules using only Scala constructs, molecules are represented as strings and parsed by a classical parsing algorithm, made of two phases (tokenization and then construction of the molecule).

3.2 Representation of reactions

The DSL makes extensive use of some special features of Scala, especially infix notations and implicit parameters. Figure 2 shows the encoding of our example problem in the DSL. Figure 3 shows a desugared version of it, revealing some aspects of the implementation.

```
equation ~ "C6H12O6" + "O2" --> "H2O" + "CO2" as reaction:
    reactants:
      200.g of "C6H12O6"
    efficiency == 60.percent
    PRINT
```

Figure 2: DSL code corresponding to the example problem.

3.2.1 Representation of equations

The syntax for equations is meant to be as close as possible to the one usually used in chemistry. Hence, we want to use `+` between molecules of the same member (left or right). This however poses an important challenge. The `+` has to be an extension method, but since molecules are represented as strings, a call to `+` on a molecule is resolved by the Scala compiler to the usual `+` method defined in the `String` class. According to the resolution rules of Scala, this method will always have precedence over extension methods. To work around this problem, equations start with the `equation` word

```
equation.~("C6H12O6").+("O2").-->("H2O").+("CO2").as(
    reaction.apply { ctx ?=>
      reactants.apply { reactantsCtx ?=>
        200.g.of("C6H12O6")(using reactantsCtx)
      }(using ctx)
      efficiency.==(60.percent)(using ctx)
      PRINT(using ctx)
    }
  )
```

Figure 3: Desugaring of the code of figure 2. References to objects are represented in orange, literals in teal, implicits in blue and cyan, and method calls in black.

(defined as an object) followed by a "~". The other functions used in the equation (`+` and `->`) have lower precedence than "~", implying that the Scala compiler will interpret the call to "~" as the method defined on the `equation` object that takes as an argument the string representation of the first reactant. This method returns an instance of the `LeftMember` class that represents the (not yet complete) left member of the equation, containing the first reactant.

The other reactants are introduced using infix calls to `+`, with a `LeftMember` as their receiver, returning a copy of the receiver extended with the molecule taken as an argument. The second member is introduced by an infix call to `->` (which, importantly, has the same precendence as `+`), returning an instance of `NoCoefEquation` that contains the now complete left member, as well as the first product. The other products are introduced by infix calls to a `+` method taking a `NoCoefEquation` as its receiver and extending it with the product given as an argument, similarly to the `+` used to build the left member.

3.2.2 Reaction parameters

Once the equation is built, one has to specify the parameters of the reaction. The DSL supports the following kinds of reaction parameters:

- A list of amounts of reactants, specifying how many moles or grams of the specified reactant(s) are available for the reaction. If omitted, the execution engine will compute the quantities based solely on the constraints on the products.
- A list of amounts of products, specifying how many moles or grams of the specified product(s) we would like to obtain. If omitted, the execution engine will compute the quantities based solely on the constraints on the reactants.
- The efficiency of the reaction.
- Optionally, a `PRINT` command to instruct the execution engine to print the computation results to the console.

These parameters appear in the block corresponding to the single argument taken by the `apply` function invoked with the `reaction` object as its receiver. The formal type of this argument is `Context ?=> Unit`, i.e. a context function mapping an implicit `Context` argument to `Unit`. The `Context` is a mutable object instantiated by `apply` and used by the methods defining the parameters, which mutate the Context according to the value of the (explicit) parameters passed to them. That is, a call to == with the efficiency object as its receiver, writes the given efficiency value to the Context. Similarly, an invocation of PRINT raises a flag inside the Context to instruct the execution engine to print the result of the computation to the console.

The case of the lists of reactants and products is slightly more complex. Indeed, in both cases, the amounts are specified by a call to of mapping a molecule to a quantity. The naive solution of implicitly passing the Context to of is not sufficient for of to know whether the specified molecule is a reactant or a product. Hence, of takes as implicit argument a MemberContext instead, which can be either a ReactantsContext or a ProductsContext, introduced by the call to apply on the reactants or products objects, respectively, using a context function in the same way as to introduce the Context. Both ReactantsContext and ProductsContext wrap the Context and implement the saveAmount method of MemberContext, but the implementation in ReactantsContext saves the amount to the list of reactants maintained by the Context, whereas the implementation in ProductsContext saves it to the list of products.

3.3 Execution

Once the equation is defined and the parameters are specified, it remains to perform the calculations and output the results. This task is devoted to the as method, which executes last as can be deduced from the precedence rules of Scala. as takes two parameters: the (not yet balanced) equation as its receiver, and the Context as its single regular parameter. The Context is returned by the call to reaction.apply, hidden behind an opaque type alias to avoid exposing it (and its mutating methods) to the user. In the case of a failure, reaction.apply returns an exception instead, which can easily be propagated to a Failure result by as.

If the retrieval of the parameters succeeds, as invokes the resolution engine by passing it the unbalanced equation and the parameters extracted from the Context. The engine will then try to balance the equation using the algorithm described in paragraph 2.3. If the balancing succeeds, then the quantities of reactants and products are computed. If the PRINT option has been specified, as also prints a summary of the results to the console.

3.4 Real-time error messages using metaprogramming

The design choice of using strings to represent molecules comes with a major drawback: the impossibility of reporting malformed molecules at compile time. To mitigate this problem, one can rely on metaprogramming. This is implemented by the parseAndStaticCheckMolecule macro, which takes as an argument a string (which has to be a literal) representing the molecule, parses it at compile time to enable early detection of malformed molecules, and replaces it with a call to the actual molecule parsing function. That way, the molecule is parsed twice: once at compile-time, for error reporting, and once at run-time, where parsing is guaranteed to succeed as the compilation would otherwise have been stopped during the expansion of the macro. It would technically be possible to get rid of this "parsing twice" behavior by expanding the macro to a programmatic representation of the molecule, but I decided against this solution because of its complexity.

The macro is implemented as an implicit function, which means that the compiler inserts it (and therefore checks the well-formedness of the molecule) whenever a string is given in a position where a molecule is expected. This is for instance the case when calling of to specify an amount of reactant or product: of expects a Molecule as its argument, but passing a string works thanks to the implicit conversion by the macro. Note that this solution does not work as well for equations: defining the `+` methods used to create equations in such a way that they expect a `Molecule` rather than a `String` as their argument results in the Scala compiler failing to resolve the method due to a confusion with the member method `+` defined in the `String` class. Hence, the `+` and `->` methods used in the syntax of equations have to invoke the macro explicitly. They are additionally defined as inline methods to make sure that the value of the string literals passed to them and representing molecules are accessible by the macro at compile-time.

4 Case studies

4.1 Output of the resolution of the motivating problem

Figure 4 shows the result obtained for the motivating example.

```
Computation succeeded:
  using reactants
    C6H12O6: 1.110 mol (200.000 g) of which 100.00% were used
    O2: 6.661 mol (213.149 g)
  according to C6H12O6 + 6 O2 --> 6 H2O + 6 CO2
  with an efficiency of 60.00%
  limiting reactant is C6H12O6
  the products are
    H2O: 3.997 mol (71.994 g)
    CO2: 3.997 mol (175.895 g)
```

Figure 4: The output of executing the resolution engine on the example problem, encoded as shown in figure 2.

We can also slightly modify the scenario and assume we have 200 grams of $C_6H_{12}O_6$ available but we only want to produce 2 moles of CO_2. We do so by adding a list of amounts of products where we write `"2.mol of "CO2""`. Figure 5 shows the encoding and output of this variant.

4.2 A more complex example: chained reactions

Let us consider the following problem statement:

25 grams of $K_2C_2O_4$ react according to $K_2C_2O_4 + KMnO_4 + H_2O \rightarrow CO_2 + Mn(OH)_2 + KOH$ to produce CO_2, which in turn reacts with NH_3 according to $CO_2 + NH_3 \rightarrow CO(NH_2)_2 + H_2O$. How much $CO(NH_2)_2$ is produced by the second reaction?

Note that the first reaction is taken from problem 1 in [2] and the second one from problem 5.2.1.7 in [1].

Encoding this problem in the DSL (see figure 6) shows how one can take advantage of the features of the host language to increase the range of problems that can be solved. Indeed, one has to query the result of the first reaction using a regular Scala call chain in order to find the amount of CO_2 reacting in the second reaction, as well as to extract the final result.

```
equation ~ "C6H12O6" + "O2" --> "H2O" + "CO2" as reaction:
    reactants:
      200.g of "C6H12O6"
    products:
      2.mol of "CO2"
    efficiency == 60.percent
    PRINT

 ----------------------------------------------------------------------

Computation succeeded:
  using reactants
    C6H12O6: 1.110 mol (200.000 g) of which 50.04% were used
    O2: 3.333 mol (106.660 g)
  according to C6H12O6 + 6 O2 --> 6 H2O + 6 CO2
  with an efficiency of 60.00%
  the products are
    H2O: 2.000 mol (36.026 g)
    CO2: 2.000 mol (88.018 g)
```

Figure 5: Encoding and output of a variant of the motivating example.

5 Conclusion and further work

The DSL described in this report is expressive enough to express simple stoichiometry problems and solve them in a reasonably user-friendly way, as showcased by the two case studies. Possible improvements include extending the support of chemical units (currently only a few units like grams and moles are supported, and most operations between them have been hard-coded based on the needs of the implementation and examples). It would make sense to explore how DSLs specialized in the handling of physical units are designed and possibly use their experience to handle units in a more systematic way in this DSL. Beyond units, the DSL would also benefit from the addition of other quantities like volumes and concentrations. On the side of the implementation, one could test the DSL more thoroughly to evaluate its user-friendliness and possibly detect error-prone aspects of the language.

```
val first =
equation~ "K2C2O4"+"KMnO4"+"H2O" --> "CO2"+"Mn(OH)2"+"KOH" as reaction:
    reactants:
        25.g of "K2C2O4"
end first

val ‘CO2‘ = first.asSuccess.amountOfProduct("CO2")

val second = equation ~ "CO2"+"NH3" --> "CO(NH2)2"+"H2O" as reaction:
    reactants:
        ‘CO2‘ of "CO2"
end second

val ‘CO(NH2)2‘ = second.asSuccess.amountOfProduct("CO(NH2)2")

// conversion from moles to grams
val mass = ‘CO(NH2)2‘ * "CO(NH2)2".mass

println(s"${mass} of CO(NH2)2")  // prints "18.065 g of CO(NH2)2"
```

Figure 6: Chain of two reactions.

References

[1] 5.2.1: Practice Problems - Reaction Stoichiometry. https://chem.libretexts.org/Courses/Oregon_Tech_PortlandMetro_Campus/OT_-_PDX_-_Metro%3A_General_Chemistry_I/05%3A_Transformations_of_Matter/5.02%3A_Reaction_Stoichiometry/5.2.01%3A_Practice_Problems-_Reaction_Stoichiometry.

[2] Stoichiometry Practice Problems. https://www.stolaf.edu/people/hansonr/chem121/Stoichiometry.pdf.