Chemical Programming: Models, Concepts, and Designs

Joshua Weinstein

2018

Contents

0.1 Preface

This book is intended to serve as a reference and guide toward programming in a chemical paradigm. It serves three main purposes. The first, and most important, is addressing the methods and models in which chemical processes can be utilized for programming. Chemical programming is an extremely young paradigm, with much to explore and create. Parts of this text will teach concepts of how to program chemically in existing programming languages.

The second is to illustrate design patterns that can be implemented in software and programming systems. Chemical programming offers an array of algorithms that can simplify common tasks such as sorting, filtering, searching and much more. The hope is to promote the development of chemical programming as a prominent paradigm in modern software development. The third, is to detail the creation of machines that can process entirely chemical programs. This includes the creation and design of a chemical virtual machine, and paths to create chemical programming languages.

This book is *not* a tutorial in how to design programs that deal with real chemical process or chemical engineering. Programming techniques for dealing with problems in chemical analysis is also not a concern of this book. Some principles of chemistry will be discussed, but not developing programs to deal with actual chemical compounds or data. The focus here is not framed around scientific computing.

Chapter 1

Introduction

Chemical Programming is a paradigm of programming that is modeled after the mechanics of chemical reactions. This style focuses on utilizing the features of chemical components, such as atoms, elements, compounds and reactions. The intention of this programming paradigm is to allow the functionality and mechanisms in chemical processes to be applied to computational values. Chemical computation permits highly customizable abstraction over traditional types like booleans, integers, and strings.

Purely modeling chemical reactions is not the goal, rather defining and modeling the components of chemistry to make them useful in programming. Computer science and chemistry are well defined, yet very different fields. The fashions and organizations of chemical reactions provide unique interfaces for program design. However, moles, molecular orbitals, and energy levels are not fruitful or applicable to abstract program designs, or a programming paradigm. Another goal in this book is to illustrate models and definitions such that components of chemicals can exist more fruitfully as a programming construct.

Traditionally, programming languages use typed values or data, such as integers, booleans, or characters. These are commonly referred to as *primitive* types. Such a type is normally the most basic level of abstraction in a programming language. They cannot be further decomposed into simpler types. The higher level abstractions such as functions or classes are composed of primitive types, which allows the creation of *user-defined* types. Similarly, elements, the most basic *type* of matter, exists in it's smallest form as an atom, which compose molecules and more sophisticated compounds.

Some fundamentals in chemistry are rather limiting when judged in a computational perspective. Reactions usually require an activation energy, E_a, in order to being and transition from reactants to products. This energy arises from a combination of heat, pressure, acidity, and other environment factors. This pushes the reactants toward a transition state, allowing the reaction to occur.

If a set of reactants does not exist in an environment or under conditions *favorable* to the products, a reaction will not occur. Similarly, a reaction can

$$A + B \xrightarrow{e} AB$$
$$B + C \rightleftharpoons BC$$

Figure 1.1: A chemical reaction with energy, and a reversible reaction

reverse itself if conditions or available energy favor the reactants. This flow of requirements and barriers does not truly relate to computational models, as restrictions in programming paradigms are self-imposed, they are not technological or physical restrictions. To adhere to the loose abstractions that a chemical program could offer, these restrictions from actual chemistry are not obeyed.

One of the main differences between chemistry and chemical programming is that reactions are *definite* transitions. A reaction can occur in a mutable or immutable state, but a reactant cannot spontaneously revert or undergo more reactions. The useful part of loose abstraction is giving a programmer more control over occurring events. *Conservation* of resources like mass, does not literally apply in chemical programming. New data is simply written to new blocks of memory in any computer, just like new objects are created in object-oriented programming languages. The return to some equilibrium or entropy is not required.

In chemistry, reactions deal with one or more molecules transforming into new molecules. As discussed in later chapters, the term *compounds* is used exclusively in this text, since a value in chemical programming does not possess real physical size. Similarly to real molecules, compounds are composed of one or more *elements*. A 32-bit integer, for example, is an element. The number zero, of any size, can also be an element. Like mentioned earlier, think of primitive types in other paradigms as equivalent to elements.

An element can be modeled in a structured or unstructured context. The structure, or *order* of the elements in a structure may or may not matter in a particular use case. Modeling for both structured or unstructured compounds can provide an array of possible reactions and computations in chemical programming. Both approaches will be described later on.

$$B_2C_2$$

$$\begin{array}{ccc} B & \text{——} & C \\ | & & | \\ B & \text{——} & C \end{array}$$

Figure 1.2: Compounds in structured versus unstructured forms

Lastly, and most importantly, the purpose in using chemical phenomena as a

backbone to programming is to allow the application of *self-organization*. Nearly any language used today only allows instructed organization of a program.

Chapter 2

Sets

In order to understand the modeling behind chemical programming, you must first be acquainted with set theory and operations that can be performed on sets. The designs and models will utilize several types of sets, including partially ordered sets and multi-sets. This chapter is a fundamental guide toward understanding set theory relevant for chemical programming, by defining key relationships and properties. It defines sets, operations like unions, intersections, complements, and more.

Sets serve as an efficient and simple representation of most concepts in chemical programming. Sets can be ordered or unordered. They can be distinct or indistinct. A set can be a subset or superset of another set. Set representation is a convenient method of expressing chemical processes in mathematical terms. Types of sets that are unique to chemical program models will also be defined and discussed in this chapter.

2.1 What is a set?

A set is a collection of one or more elements. It is usually denoted with curly braces, {}. A set's elements can be expressed as some series of E_n elements.

$$S = \{E_n \dots E_k\}$$

If an element, E is contained in a set S, then,

$$E \in S$$

and if a set S has only one member, element E, then,

$$E \equiv S,$$
$$E \equiv \{E\}$$

A set can be contained within some other set. If this is true, a set is a subset of another set. Let us have one set $s = \{a, b\}$, and another set, $S = \{a, b, c\}$.

$$\forall x\{\exists x \in s \rightarrow \exists x \in S\}$$

where x is an element, this can be described more conveniently as

$$s \subseteq S$$

A set that has no elements is called an empty set, denoted $\emptyset$. A set is an empty set when,

$$\forall x \rightarrow \neg\exists x \in s$$

If an element is not in some set, this is written as $E \notin S$. A distinct element, E_i can have multiple set memberships. $E_i \in S_1 \wedge E_i \in S_2$ implies a membership in the first and second set. This is also called having membership in the intersection of both sets.

2.1.1 Intersection

An intersection is a set that is composed of the elements present in two or more sets. It is a separate set expressed as the elements present in both some set S_a and S_b. It is denoted with a $\cap$, such that,

$$S_a \cap S_b = \{x \mid x \in S_a \wedge x \in S_b\}$$

For the intersection of an arbitrary number of sets, or for a collection, C, whose elements are entirely sets, we can write

$$\left\{x \in \bigcap_{S \in C} S\right\}$$

Intersections are also a way in which two sets' equality can be determined. If the intersection of two sets is an empty set, then those two sets share no elements and are *completely* unequal, $S_a \cap S_b = \emptyset$. Conversely, if the set of elements in one set OR the other set is equivalent to the intersection of those two sets, the sets are equal to each other. Elements that reside in either one set or another is called a *union*.

2.1.2 Unions

A union is a set that represents the elements of two or more sets. It is denoted with a $\cup$ Specifically,

$$S_a \cup S_b = \{x \mid x \in S_a \vee x \in S_b\}$$

The union, in the case of two sets, S_a and S_b , is composed of three *distinct* sets. The third set, aside from the elements in S_a or S_b is the intersection,

$S_a \cap S_b$. The union, in this form, is expressed as $S_a \cup S_b = \{S_a, S_b, (S_a \cap S_b)\}$. The membership of S_a and S_b and the exclusion from the intersection is called the symmetric difference of sets. Symmetric difference is the opposite of intersections, it judges how *unlike* two sets are.

2.1.3 Equality and Difference

Two sets are equal if and only if their intersection is equal to their union.

$$S_a = S_b \iff S_a \cap S_b = S_a \cup S_b$$

This is true since the union of two sets is the set of all members across both of them, if every member of either set is in both sets, there is no such member that exists in one set and not the other. This idea can also be used to define set inequality.

A set S_a is not equal to a set S_b when at least one element is not present in both sets,

$$S_a \neq S_b \iff \exists x \in S_a, x \notin S_b \tag{2.1}$$
$$S_a \neq S_b \iff \exists x \in S_a \cup S_b, x \notin S_a \cap S_b \tag{2.2}$$

The set of elements in some set S_a *or* S_b but not both is the symmetric difference of the sets, denoted with a $\triangle$

$$S_a \triangle S_b = \{x \mid x \notin (S_a \cap S_b) \wedge x \in S_a \vee x \in S_b\}$$

To relate the idea of set equality and difference, the intersection of two sets and their symmetric difference are *opposites*. A ratio can be established between the count of two sets as a degree of equality, Q_e.

$$Q_e = \frac{S_a \cap S_b}{S_a \triangle S_b}$$

If Q_e is 1, there are no members in the symmetric difference, and every member of the union is also a member of the intersection. If Q_e is 0, this means there are no members in the intersection, and every member of the union is a member of the symmetric difference. The varying degrees between 1 or 0 can determine how similar a set is to another.

2.2 Multisets

Previously, we discussed sets with distinct collections of elements, such that for every unique member, only one of that member is present in the set. A basic set cannot contain multiplicities of it's members, while a multiset can. All basic sets can be expressed as multisets where each member has a multiplicity of 1. If $S = \{E_1 \dots E_n\}$ is a basic set, then a multiset can be expressed as:

$$M = \left\{ E_1^{m(e_1)}, \ldots E_n^{m(e_n)} \right\}$$

where $m(e)$ is the multiplicity of some element e in the set. An example multiset can also be written more simply as $\{a^2, b^1\}$, which contains two of the element a and one of the element b. There are many varieties of multisets, as even sets with the same types of elements can carry different multiplicities. Multisets are particularly important in chemical programming as they mirror the representation of a chemical compound, such as water, H_2O. A molecule of water can be expressed as a multiset with two hydrogen atoms and one oxygen atom.

2.2.1 Membership

With basic sets, membership is defined as a simple boolean value, whether or not a set contains some element,

$$x \in S$$

In a multiset, membership is an integer value, where 0 indicates no membership, and 1 or more corresponds to the element's multiplicity.

$$x \in M \iff \exists x : m(x) \geq 1$$

Conversely, not being in the multiset is defined as the multiplicity being zero, $x \notin M \iff m(x) = 0$. The multiplicity of an element also determines how much of the set it represents. An element with the highest multiplicity in the set is the most common element. An element is the *mode* of the multiset if,

$$x = mode(M) \iff \forall e \in M \rightarrow m(e) < m(x)$$

When all the members of a multiset have the same multiplicity, there is no mode. The judgement of membership based on a numeric value rather than a boolean value can also be expressed as a function, This function's domain is any possible member, and it's range is the multiplicity of the input element,

$$M_f : e \rightarrow m$$

An *empty* multiset would always have a range of zero, such that,

$$\emptyset = M_f : e \rightarrow 0$$

However, for the purposes of chemical programming, multisets are best represented as collections rather than functions.

2.2.2 Operations

Multisets can undergo several operations that basic sets cannot. Basic set operations and their definitions such as a union or an intersection don't apply well to multisets, due to the absence of the idea of multiplicity. Such as with equality, two multisets could have the same members yet different multiplicities of each member.

The first operation of multisets is addition. The addition operation results in a new set where the multiplicities and variety of each member of both multisets are placed into one. The operation is binary and denoted with a $\uplus$.

$$M_1 \uplus M_2 = \{x, y \mid (x, m(x)) \in M_1, (y, m(y)) \in M_2\}$$

For example, if $M_1 = \{a, a, b\}$, and $M_2 = \{c, c, b\}$, the addition set is, $\{a, a, b, b, c, c\}$. Multisets can also be subtracted from one another. The subtraction of two multisets mirrors the symmetric difference of two basic sets. It is a multiset of the elements, that do not have equal multiplicities between some multiset M_1 and multiset M_2.

$$M_1 - M_2 = \{(x, m(x)) \mid m_1(x) - m_2(x) \neq 0 \rightarrow x \in M_1, x \in M_2\}$$

Two multisets are equal to each other if and only if their difference set is the empty multiset.

$$M_1 = M_2 \iff M_1 - M_2 = \emptyset$$

Similar to basic sets, the size of the difference set indicates how alike or different two multisets are. The larger the difference set, the more elements that M_1 and M_2 *do not* share or have the same number of. The smaller the difference set, the more equal multiplicities shared between M_1 and M_2.

2.3 Nested Sets

Sets can contain members that are also sets themselves. Such a set that is contained within another set can be treated as a unique, distinct value, as opposed to a collection. Normally, sets are not regarded singularly. The idea of a subset, such as $S_1 \subseteq S_2$ reads that all the elements in S_1 are also in S_2. Yet this does not handle the case of a set with all the same members as S_1 being contained in S_2. For basic set membership, the following would hold true:

```
int c = 0;
char g = 'g';
```

Chapter 3

Compounds

Compounds are foo.