Table of Contents

`Chymyst`: declarative concurrency in Scala

`Chymyst` is a framework for concurrency in functional programming implementing the **chemical machine** paradigm, also known in the academic world as Join Calculus. The chemical machine concurrency paradigm has the same expressive power as CSP (Communicating Sequential Processes) or the Actor model.

`Chymyst Core` is a library that implements the high-level concurrency primitives as a domain-specific language in Scala. `Chymyst` is a framework-in-planning that will build upon `Chymyst Core` and bring declarative concurrency to practical applications.

The code of `Chymyst Core` is based on previous Join Calculus implementations by He Jiansen (https://github.com/Jiansen/ScalaJoin, 2011) and Philipp Haller (http://lampwww.epfl.ch/~phaller/joins/index.html, 2008), as well as on my earlier prototypes in Objective-C/iOS and Java/Android.

The *Concurrency in Reactions* tutorial book: table of contents

Overview of `Chymyst` and the chemical machine paradigm

Concurrency in Reactions: Get started with this extensive tutorial book

From actors to reactions: a guide for those familiar with the Actor model

A "Hello, world" project

Presentations on `Chymyst` and the chemical machine programming paradigm

Oct. 16, 2017: Talk given at the Scala Bay meetup:

- Talk slides with audio
- See also the talk slides (PDF) and the code examples for the talk.

July 2017: Draft of an academic paper describing Chymyst and its approach to join calculus

Nov. 11, 2016: Talk given at Scalæ by the Bay 2016:

- Video presentation of early version of `Chymyst Core`, then called `JoinRun`
- See also the talk slides revised for the current syntax.

Main features of the chemical machine

Comparison of the chemical machine vs. academic Join Calculus

Comparison of the chemical machine vs. the Actor model

Comparison of the chemical machine vs. the coroutines / channels approach (CSP)

Technical documentation for `Chymyst Core`

Source code repository for `Chymyst Core`

Version history and roadmap

Status

The `Chymyst Core` library is in alpha pre-release, with very few API changes envisioned for the future.

The semantics of the chemical machine (restricted to single-host, multicore computations) is fully implemented and tested on many nontrivial examples.

The library JAR is published to Maven Central.

Extensive tutorial and usage documentation is available.

Unit tests include examples such as asynchronous counter, parallel “or”, concurrent merge-sort, and “dining philosophers”. Test coverage is 100% according to codecov.io.

Performance benchmarks indicate that `Chymyst Core` can schedule about 100,000 reactions per second per CPU core, and the performance bottleneck is in submitting jobs to threads (a distant second bottleneck is pattern-matching in the internals of the library).

Preface

`Chymyst` is a library and a framework for declarative concurrent programming.

It follows the **chemical machine** paradigm (known in the academic world as “Join Calculus”) and is implemented as an embedded DSL (domain-specific language) in Scala.

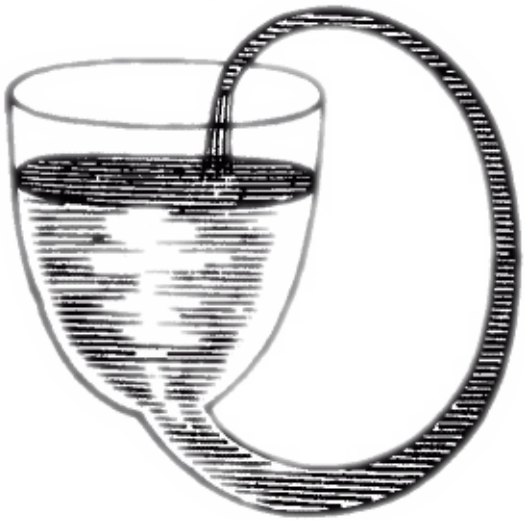
The goal of this tutorial is to explain the chemical machine paradigm and to show examples of implementing concurrent programs in `Chymyst`. To understand this tutorial, the reader should have some familiarity with the `Scala` programming language.

Source code

The source code repository for `Chymyst Core` is at https://github.com/Chymyst/chymyst-core.

Although this tutorial focuses on using `Chymyst` in the Scala programming language, one can straightforwardly embed the chemical machine as a library on top of any programming language that has threads and semaphores. The main concepts and techniques of the chemical machine paradigm are independent of the chosen programming language. However, a purely functional language is a better fit for the chemical machine.

Dedication to Robert Boyle (1626-1691)



This drawing is by Robert Boyle, who was one of the founders of the science of chemistry. In 1661 he published a treatise titled *“The Sceptical Chymyst”*, from which the `Chymyst` framework borrows its name.

Quick start

`Chymyst Core` implements a declarative DSL for purely functional concurrency in Scala. The DSL is based on the "chemical machine" paradigm, which is likely unfamiliar to most readers.

This chapter is for the impatient readers who want to dive straight into the code, with very few explanations.

Read the next chapter if you prefer to understand the concepts before looking at code.

Setup

First, declare this library dependency in your `build.sbt`:

```
libraryDependencies += "io.chymyst" %% "chymyst-core" % "latest.integration"
```

The `Chymyst Core` DSL becomes available once you add this statement:

```
scala> import io.chymyst.jc._
import io.chymyst.jc._
```

This imports all the necessary symbols such as `m`, `b`, `site`, `go` and so on.

Concurrent programming: processes and data

In the chemical machine, an asynchronous concurrent process (called a **reaction**) is implemented as a computation that works with a special kind of data called **molecules**. A reaction can consume one or more input molecules and may emit (zero or more) new molecules.

Molecules are created out of ordinary data values by calling special **molecule emitters**.

All molecule emitters must be declared before using them. A new molecule emitter is created using the special syntax `m[T]`, where `T` is the type of the value:

```
scala> val in = m[Int] // emitter for molecule `in` with value of type `Int`
in: io.chymyst.jc.M[Int] = in

scala> val result = m[Int] // emitter for molecule `result` with value of type `String`

result: io.chymyst.jc.M[Int] = result
```

Molecules can be emitted using this syntax:

```
val c = m[Int] // emitter for molecule `c` with value of type `Int`
c(123) // emit a new molecule `c()` carrying the `Int` value `123`
```

A reaction must be declared using the `go { }` syntax. The body of a reaction is a computation that can contain arbitrary Scala code.

In order to activate one or more reactions, use the `site()` call.

```
scala> site(
     |   go { case in(x) ⇒     // consume a molecule `in(...)` as input
     |   // now declare the body of the reaction:
     |     val z = x * 2       // compute some new value using the value `x`
     |     result(z)           // emit a new mmolecule `result(z)`
     |   },
     |   go { case result(x) ⇒ println(x) } // consume `result(...)`
     | )
res0: io.chymyst.jc.WarningsAndErrors = In Site{in → ...; result → ...}: no warnings o
r errors

scala> in(123); in(124); in(125)   // emit some initial molecules

scala> Thread.sleep(200) // wait for reactions to start
250
248
246
```

Emitters can be called many times to emit many copies of a molecule:

```
in(123); in(124); in(125)
(1 to 10).foreach(x ⇒ in(x))
```

All emitted molecules become available for reactions to consume them. Reactions will start in parallel whenever the required input molecules are available.

A reaction can depend on *several* input molecules at once, and may emit several molecules as output. The actual computation will start only when *all* its input molecules are available (have been emitted and not yet consumed by other reactions).

```
scala> val in1 = m[Int] // molecule `in1`
in1: io.chymyst.jc.M[Int] = in1

scala> val in2 = m[Int] // molecule `in2`
in2: io.chymyst.jc.M[Int] = in2

scala> val result = m[Boolean] // molecule `result` with value of type `Boolean`
result: io.chymyst.jc.M[Boolean] = result

scala> site(
     |   go { case in1(x) + in2(y) ⇒   // wait for two molecules
     |     println(s"got x = $x, y = $y")  // debug output
     |     val z: Boolean = x != y // compute some new value `z`
     |     result(z) // emit `result` molecule with value `z`
     |     val t: Boolean = x > y // another computation, whatever
     |     result(t) // emit another `result` molecule
     |     println(s"emitted result($z) and result($t)")
     |   },
     |   go { case result(x) ⇒ println(s"got result = $x") }
     | )
res3: io.chymyst.jc.WarningsAndErrors = In Site{in1 + in2 → ...; result → ...}: no war
nings or errors

scala> in2(20)

scala> in1(10) // emit initial molecules

scala> Thread.sleep(200) // wait for reactions to run
got x = 10, y = 20
emitted result(true) and result(false)
got result = true
got result = false
```

Emitting a molecule is a *non-blocking* operation; execution continues immediately, without waiting for any reactions to start. Reactions will start as soon as possible and will run in parallel with the processes that emitted their input molecules.

Molecules can carry data of any type as their **payload value** (but the type is fixed by the declared emitter's type). For example, a molecule can carry a payload value of function type, which allows us to implement **asynchronous continuations**:

```
scala> val in = m[Int] // input molecule
in: io.chymyst.jc.M[Int] = in

scala> val cont = m[Int ⇒ Unit]  // molecule that carries the continuation
cont: io.chymyst.jc.M[Int => Unit] = cont

scala> site(
     |   go { case in(x) + cont(k) ⇒
     |     println(s"got x = $x")
     |     val z : Int = x * x   // compute some output value
     |     k(z) // invoke continuation
     |   }
     | )
res7: io.chymyst.jc.WarningsAndErrors = In Site{cont + in → ...}: no warnings or error
s

scala> in(100) // emit initial molecule

scala> // emit the second molecule required by reaction
     | cont(i ⇒ println(s"computed result = $i"))

scala> Thread.sleep(200)
got x = 100
computed result = 10000
```

New reactions and molecules can be defined anywhere in the code, for instance, within a function scope or within the local scope of another reaction's body.

Example: Asynchronous counter

Non-blocking read access

We implement a counter that can be incremented and whose value can be read. Both the increment and the read operations are asynchronous (non-blocking). The read operation is implemented as an asynchronous continuation.

```
scala> val counter = m[Int]
counter: io.chymyst.jc.M[Int] = counter

scala> val incr = m[Unit] // `increment` operation
incr: io.chymyst.jc.M[Unit] = incr

scala> val read = m[Int ⇒ Unit] // continuation for the `read` operation
read: io.chymyst.jc.M[Int => Unit] = read

scala> site(
     |   go { case counter(x) + incr(_) ⇒ counter(x + 1) },
     |   go { case counter(x) + read(cont) ⇒
     |     counter(x) // Emit the `counter` molecule with unchanged value `x`.
     |     cont(x) // Invoke continuation.
     |   }
     | )
res12: io.chymyst.jc.WarningsAndErrors = In Site{counter + incr → ...; counter + read
→ ...}: no warnings or errors

scala> counter(0) // Set initial value of `counter` to 0.

scala> incr() // Short syntax: emit a molecule with a `Unit` value.

scala> incr() // This can be called from any concurrently running code.

scala> read(i ⇒ println(s"counter = $i")) // this too

scala> Thread.sleep(200)
counter = 2
```

A molecule can be consumed only by *one* instance of a reaction. For this reason, there is no race condition when running this program, even if several copies of the molecules `incr()` and `read()` are emitted from several concurrent processes.

Non-blocking wait until done

We now implement a counter that is incremented until some condition is met. At that point, we would like to start another computation that uses the last obtained counter value.

```
scala> val counter = m[Int]
counter: io.chymyst.jc.M[Int] = counter

scala> val done = m[Int] // Signal the end of counting.
done: io.chymyst.jc.M[Int] = done

scala> val next = m[Int ⇒ Unit] // continuation
next: io.chymyst.jc.M[Int => Unit] = next

scala> val incr = m[Unit] // `increment` operation
incr: io.chymyst.jc.M[Unit] = incr

scala>  // The condition we are waiting for, for example:
     | def areWeDone(x: Int): Boolean = x > 1
areWeDone: (x: Int)Boolean

scala> site(
     |   go { case counter(x) + incr(_) ⇒
     |     val newX = x + 1
     |     if (areWeDone(newX)) done(newX)
     |     else counter(newX)
     |   },
     |   go { case done(x) + next(cont) ⇒
     |     cont(x) // invoke continuation on the value `x`
     |   }
     | )
res19: io.chymyst.jc.WarningsAndErrors = In Site{counter + incr → ...; done + next → .
..}: no warnings or errors

scala> counter(0) // set initial value of `counter` to 0

scala> incr() // Emit a molecule with `Unit` value.

scala> incr() // This can be called from any concurrent process.

scala> next { x ⇒
     | // Continue the computation, having obtained `x`.
     |   println(s"counter = $x")
     | // more code...
     | }

scala> Thread.sleep(200)
counter = 2
```

More code can follow `println()`, but it will be constrained to the scope of the closure under `next()`.

Blocking emitters

In the previous example, we used a continuation in order to wait until some condition is satisfied. `Chymyst` implements this often-used pattern via special emitters called **blocking emitters**. Using this feature, the previous code can be rewritten more concisely:

```
scala> val counter = m[Int]
counter: io.chymyst.jc.M[Int] = counter

scala> val done = m[Int] // signal the end of counting
done: io.chymyst.jc.M[Int] = done

scala> val next = b[Unit, Int] // blocking emitter with integer reply value
next: io.chymyst.jc.B[Unit,Int] = next/B

scala> val incr = m[Unit] // `increment` operation
incr: io.chymyst.jc.M[Unit] = incr

scala>  // the condition we are waiting for, for example:
     | def areWeDone(x: Int): Boolean = x > 1
areWeDone: (x: Int)Boolean

scala> site(
     |   go { case counter(x) + incr(_) ⇒
     |     val newX = x + 1
     |     if (areWeDone(newX)) done(newX)
     |     else counter(newX)
     |   },
     |   go { case done(x) + next(_, reply) ⇒
     |     reply(x) // emit reply with integer value `x`
     |   }
     | )
res26: io.chymyst.jc.WarningsAndErrors = In Site{counter + incr → ...; done + next/B →
 ...}: no warnings or errors

scala> counter(0) // set initial value of `counter` to 0

scala> incr() + incr() // same as `incr(); incr()`

scala> val x = next() // block until reply is sent
x: Int = 2
```

More code can follow `println()`, and that code can use `x` and is no longer constrained to the scope of a closure, as before.

Blocking emitters are declared using the `b[T, R]` syntax, where `T` is the type of the molecule's payload value and `R` is the type of their **reply value**.

Asynchronous counter: blocking read access

We can use a blocking emiter to implement blocking access to the counter's current value.

```
scala> val counter = m[Int]
counter: io.chymyst.jc.M[Int] = counter

scala> val read = b[Unit, Int] // `read` is a blocking emitter
read: io.chymyst.jc.B[Unit,Int] = read/B

scala> val incr = m[Unit] // `increment` operation is asynchronous
incr: io.chymyst.jc.M[Unit] = incr

scala> site(
     |   go { case counter(x) + incr(_) ⇒ counter(x + 1) },
     |   go { case counter(x) + read(_, reply) ⇒
     |     counter(x) // emit x again as the payload value on the `counter` molecule
     |     reply(x) // emit reply with value `x`
     |   }
     | )
res29: io.chymyst.jc.WarningsAndErrors = In Site{counter + incr → ...; counter + read/B
 → ...}: no warnings or errors

scala> counter(0) // set initial value of `counter` to 0

scala> incr()

scala> incr() // these emitter calls do not block

scala> val x = read() // block until reply is sent
x: Int = 2
```

Parallel `map`

We now implement a parallel `map` operation: apply a function to every element of a list, and produce a list of results.

An asynchronous counter is used to keep track of progress. For simplicity, we will aggregate results into the final list in the order they are computed. The molecule called `done()` is emitted when the entire list is processed. Also, a blocking emitter `waitDone` is used to wait for the completion of the job.

```
scala> val start = m[Int] // molecule value is a list element
start: io.chymyst.jc.M[Int] = start

scala> def f(x: Int): Int = x * x // some computation
f: (x: Int)Int

scala> val total = 10
total: Int = 10

scala> val counter = m[Int]
counter: io.chymyst.jc.M[Int] = counter

scala> val incr = m[Unit]
incr: io.chymyst.jc.M[Unit] = incr

scala> val result = m[List[Int]]
result: io.chymyst.jc.M[List[Int]] = result

scala> val done = m[Unit] // signal the end of computation
done: io.chymyst.jc.M[Unit] = done

scala> val waitDone = b[Unit, List[Int]] // blocking emitter
waitDone: io.chymyst.jc.B[Unit,List[Int]] = waitDone/B

scala> site(
     |   go { case start(i) + result(xs) ⇒
     |     val newXs = f(i) :: xs // compute i-th element concurrently and append
     |     result(newXs)
     |     incr()
     |   },
     |   go { case incr(_) + counter(n) ⇒
     |     val newN = n + 1
     |     if (newN == total) done()
     |     else counter(newN)
     |   },
     |   go { case done(_) + waitDone(_, reply) + result(xs) ⇒ reply(xs) }
     | )
res33: io.chymyst.jc.WarningsAndErrors = In Site{counter + incr → ...; done + result +
 waitDone/B → ...; result + start → ...}: no warnings or errors

scala> // emit initial values
     | (1 to total).foreach(i ⇒ start(i))

scala> counter(0)

scala> result(Nil)

scala> waitDone() // block until done, get result
res38: List[Int] = List(100, 81, 64, 49, 36, 25, 16, 9, 4, 1)
```

The chemical machine paradigm

`Chymyst` adopts an unusual approach to concurrent programming. This approach does not use threads, actors, futures, or monads. Instead, concurrent computations are performed by a special runtime engine that simulates chemical reactions. This approach can be more easily understood by first considering the **chemical machine** metaphor.

Simulation of chemical reactions

Imagine that we have a large tank of water where many different chemical substances are dissolved. Different chemical reactions are possible in this “chemical soup”, as various molecules come together and react, producing other molecules. Reactions could start at the same time (i.e. concurrently) in different regions of the soup.

Chemical reactions are written like this:

$HCl + NaOH \Rightarrow NaCl + H_2O$

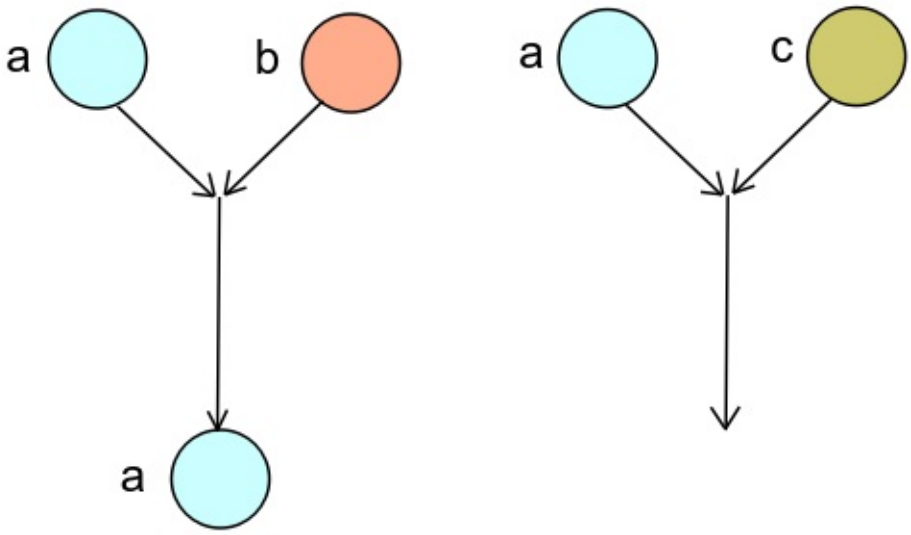
A molecule of hydrochloric acid (HCl) reacts with a molecule of sodium hydroxide (NaOH) and yields a molecule of salt (NaCl) and a molecule of water (H_2O).

Since we are going to simulate reactions in a computer, we make the “chemistry” completely arbitrary. We can define molecules of any sort, and we can postulate arbitrary reactions between them.

For instance, we can postulate that there exist three sorts of molecules called `a`, `b`, `c`, and that they can react as follows:

`a + b → a`

`a + c →` [*nothing*]



Of course, real-life chemistry does not allow a molecule to disappear without producing any other molecules. But our chemistry is purely imaginary, and so the programmer is free to postulate arbitrary chemical laws.

To develop the chemical analogy further, we allow the chemical soup to hold many copies of each molecule. For example, the soup can contain five hundred copies of `a` and three hundred copies of `b`, and so on. We also assume that we can emit any molecule into the soup at any time.

It is not difficult to implement a simulator for the chemical behavior we just described. Having specified the list of chemical laws and emitted some initial molecules into the soup, we start the simulation. The chemical machine will run all the reactions that are allowed by the chemical laws.

We will say that in a reaction such as

```
a + b + c → d + e
```

the **input molecules** are `a`, `b`, and `c`, and the **output molecules** are `d` and `e`. A reaction can have one or more input molecules, and zero or more output molecules.

When a reaction starts, the input molecules instantaneously disappear from the soup (we say they are **consumed** by the reaction), and then the output molecules are **emitted** into the soup.

The simulator will start many reactions concurrently whenever their input molecules are available. As reactions emit new molecules into the soup, the simulator will continue starting new reactions whenever possible.

Concurrent computations using the chemical machine

The simulator described in the previous section is at the core of the runtime engine of `Chymyst`. Rather than merely watch as reactions happen, we are going to use the chemical machine to perform actual computations. To this end, the following features are added:

1. Each molecule in the soup is required to *carry a value*. Molecule values are strongly typed: a molecule of a given sort (such as `a` or `b`) can only carry values of some fixed type (such as `Boolean` or `String`). The programmer is completely free to specify what kinds of molecules are defined and what types they carry.
2. Since molecules must carry values, we need to specify a value of the correct type whenever we emit a new molecule into the soup.
3. For the same reason, reactions that emit new molecules will need to put values on each of the output molecules. These output values must be *functions of the input values*, — that is, of the values carried by the input molecules consumed by this reaction. Therefore, each chemical reaction must carry a Scala expression (called the **reaction body**) that will compute the new output values and emit the output molecules.

With these three additional features, the simple chemical simulator becomes what is called in the academic literature a "reflexive chemical abstract machine". We will call it, for short, the **chemical machine**.

These three features are essentially a complete description of the chemical machine paradigm. The rest of this book is a guide to programming the chemical machine. We will use simple logical reasoning to figure out what reactions are necessary for various concurrent computations.

Reactions in pseudo-code

We will use syntax such as `b(123)` to denote molecules. In a chemical reaction, the syntax `b(123)` means that the molecule `b` carries an integer value `123`. Molecules to the left-hand side of the arrow are the input molecules of the reaction; molecules on the right-hand side are the output molecules.

A typical reaction, equipped with molecule values and a reaction body, looks like this in pseudo-code syntax:

```
a(x) + b(y) → a(z)
  where z = computeZ(x, y)
```

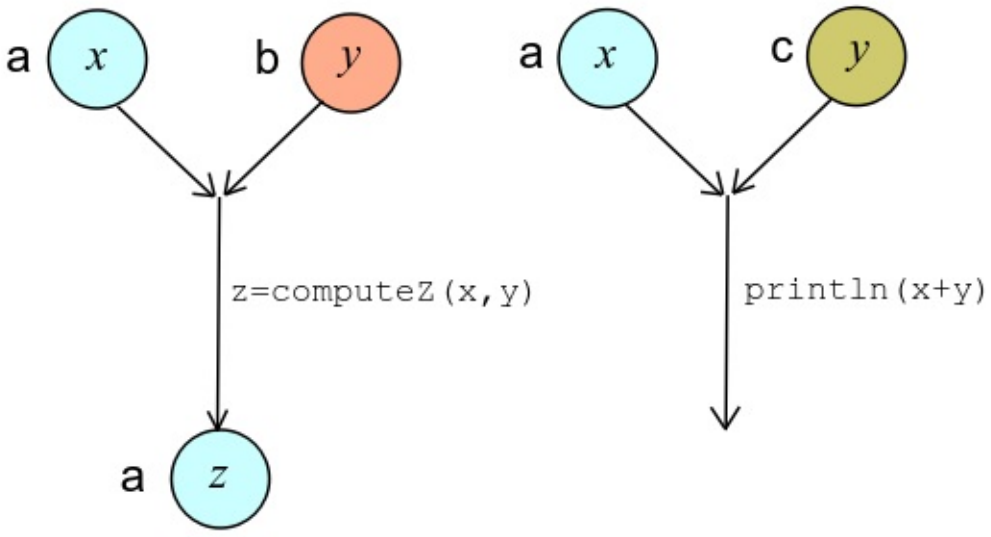
In this example, the reaction's input molecules are `a(x)` and `b(y)`; that is, the input molecules have chemical designations `a` and `b` and carry values `x` and `y` respectively.

The reaction body is an expression that captures the values `x` and `y` from the consumed input molecules. The reaction body computes `z` out of `x` and `y`; in this example, this is done using the function `computeZ`. The newly computed value `z` is placed onto the output molecule `a(z)`, which is emitted back into the soup.

Another example of a reaction is

```
a(x) + c(y) → println(x + y) // reaction body with no output molecules
```

This reaction consumes the molecules `a` and `c` as its input, but does not emit any output molecules. The only result of running the reaction is the side-effect of printing the number `x + y`.



The computations performed by the chemical machine are *automatically concurrent*: Whenever input molecules are available in the soup, the runtime engine will start a reaction that consumes these input molecules. If many copies of input molecules are available, the

runtime engine could start several reactions concurrently. The runtime engine will decide how many concurrent reactions to run, depending on the number of available cores or other considerations.

The reaction body can be a *pure function* that computes output values solely from the input values carried by the input molecules. If the reaction body is a pure function, it is completely safe (free of contention or race conditions) to execute concurrently several copies of the same reaction. Each copy of the reaction will run in its own process, consuming its own set of input molecules and working with its own input values. This is how the chemical machine achieves safe and automatic concurrency in a purely functional way, with no shared mutable state.

The syntax of `Chymyst`

So far, we have been writing chemical laws in pseudo-code. The actual syntax of `Chymyst` is only a little more verbose. Reactions are defined using `case` expressions that specify the input molecules by pattern-matching.

Here is the translation of the example reaction shown above into the syntax of `Chymyst`:

```
import io.chymyst.jc._

// declare the molecule types
val a = m[Int] // a(...) will be a molecule with an integer value
val b = m[Int] // ditto for b(...)

// declare the reaction site and the available reaction(s)
site(
  go { case a(x) + b(y) ⇒
    val z = computeZ(x,y)
    a(z)
  }
)
```

The helper functions `m`, `site`, and `go` are defined in the `Chymyst` library.

The function `go(...)` defines a reaction. The left-hand side of a reaction is the set of its input molecules, represented as a pattern-matching expression `case a(x) + ...`. The values of the input molecules are pattern variables. The right-hand side of a reaction is an arbitrary Scala expression that can use the pattern variables to compute new values.

The function call `site(...)` declares a **reaction site**, which can be visualized as a place where molecules gather and wait for their reaction partners.

Example: Asynchronous counter

We already know enough to start implementing our first concurrent program!

The task at hand is to maintain a counter with an integer value, which can be incremented or decremented by non-blocking (asynchronous) requests. It should be safe to increment and decrement the counter from different processes running at the same time.

To implement this in `Chymyst`, we begin by deciding which molecules we will need to use. Since the chemical machine paradigm does not use shared mutable state, it is clear that the integer value of the counter needs to be carried by a molecule. Let's call this molecule `counter` and specify that it carries an integer value:

```
val counter = m[Int]
```

The increment and decrement requests must be represented by other molecules. Let us call them `incr` and `decr`. These molecules do not need to carry values, so we will define the `Unit` type as their value type:

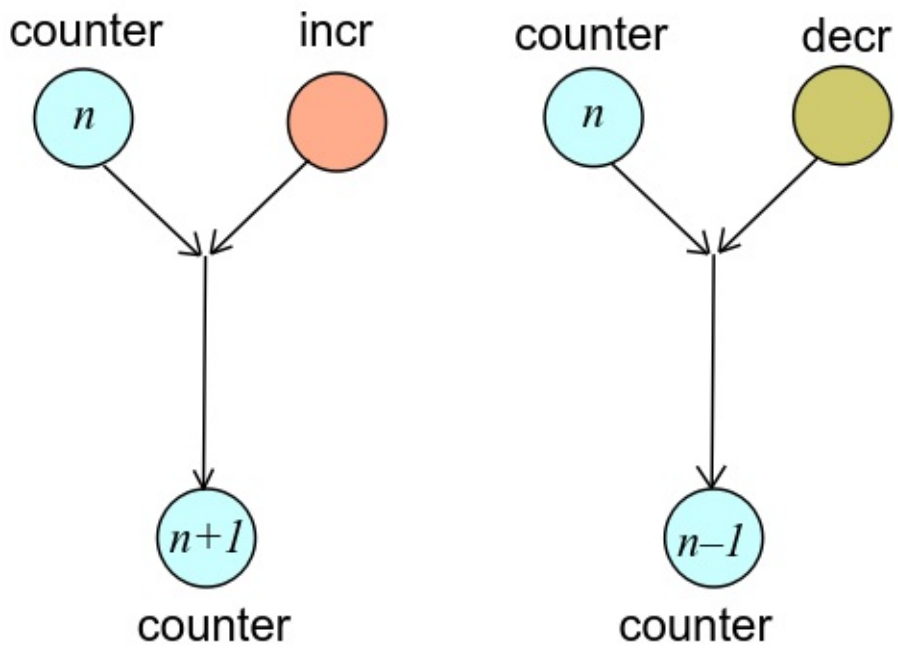
```
val incr = m[Unit]
val decr = m[Unit]
```

Now we need to define the chemical reactions. The reactions must be such that the counter's value is incremented when we emit the `incr()` molecule, and decremented when we emit the `decr()` molecule.

So, it looks like we will need two reactions. Let us create a reaction site:

```
site(
  go { case counter(n) + incr(_) ⇒ counter(n + 1) },
  go { case counter(n) + decr(_) ⇒ counter(n - 1) }
)
```

Each reaction says that the new value of the counter (either `n + 1` or `n - 1`) will be carried by the new `counter(...)` molecule emitted by the reaction's body. The previous molecule, `counter(n)`, will be consumed by the reactions. The `incr()` and `decr()` molecules will be likewise consumed.



In `Chymyst`, a reaction site can declare one or more reactions, since the function `site()` takes a variable number of arguments.

In the present example, however, both reactions need to be written within the same reaction site. Here is why:

Both reactions `counter + incr → ...` and `counter + decr → ...` consume the molecule `counter()`. In order for any of these reactions to start, the molecule `counter()` needs to be present at some reaction site. Therefore, the molecules `incr()` and `decr()` must be present at the *same* reaction site, or else they cannot meet with `counter()` to start a reaction. For this reason, both reactions need to be defined *together* in a single reaction site.

After defining the molecules and their reactions, we can start emitting new molecules into the soup:

```
counter(100)
incr() // after a reaction with this, the soup will have counter(101)
decr() // after a reaction with this, the soup will have counter(100)
decr() + decr() // after a reaction with these, the soup will have counter(98)
```

The syntax `decr() + decr()` is just a chemistry-resembling syntactic sugar for emitting several molecules at once. The expression `decr() + decr()` is equivalent to `decr(); decr()`.

Note that `counter`, `incr` and `decr` are local values that we use as functions, e.g. by writing `counter(100)` and `decr()`, when we need to emit the corresponding molecules. For this reason, we refer to `counter`, `incr`, and `decr` as **molecule emitters**.

It could happen that we are emitting `incr()` and `decr()` molecules too quickly for reactions to start. This will result in many instances of `incr()` or `decr()` molecules being present in the soup, waiting to be consumed. Is this a problem?

Recall that when the chemical machine starts a reaction, all input molecules are consumed first, and only then the reaction body is evaluated. In our case, each reaction needs to consume a `counter()` molecule, but only one instance of `counter()` molecule is initially present in the soup. For this reason, the chemical machine will need to choose whether the single `counter()` molecule will react with an `incr()` or a `decr()` molecule. Only when the incrementing or the decrementing calculation is finished, the new instance of the `counter()` molecule (with the updated integer value) will be emitted into the soup. This automatically prevents race conditions with the counter: There is no possibility of updating the counter value simultaneously from different reactions.

Tracing the output

The code shown above will not print any output, so it is instructive to put some print statements into the reaction bodies.

```
import io.chymyst.jc._

// declare the molecule emitters and the value types
val counter = m[Int]
val incr = m[Unit]
val decr = m[Unit]

// helper function to be used in reactions
def printAndEmit(x: Int) = {
  println(s"new value is $x")
  counter(x)
}

// write the reaction site
site(
  go { case counter(n) + decr(_) ⇒ printAndEmit(n - 1) }
  go { case counter(n) + incr(_) ⇒ printAndEmit(n + 1) },
)

counter(100)
incr() // prints “new value is 101”
decr() // prints “new value is 100”
decr() + decr() // prints “new value is 99” and then “new value is 98”
```

Exercises

Producer-consumer

Implement a chemical program that simulates a simple “producer-consumer” arrangement.

There exist one or more items that can be consumed. Each item is labeled by an integer value. Any process can issue an (asynchronous, concurrent) request to consume an item. If there is an item available, it should be consumed and its label value printed to the console. If no items are available, the consume request should wait until items become available.

Initially, there should be `n` items present, labeled by integer values 1 to `n`.

Your code should attempt to consume three items.

The program should define a molecule `item()` carrying an integer value, a molecule `consume()` carrying a unit value, and an appropriate reaction.

(At this point, your chemical program does not need to be able to stop. It is sufficient that the correct values are printed to the console.)

Solution

```
val item = m[Int]
val consume = m[Unit]
val n = 10 // or `n` could be a runtime parameter
site(
  go { case item(x) + consume(_) ⇒ println(s"consumed $x") }
)
(1 to n).foreach(i ⇒ item(i))

consume() + consume() + consume()
```

Batch producer

Modify the previous program, adding a request to produce a new set of `n` items, again labeled with values 1 to `n`.

This request should be modeled by a new molecule `produce()` with unit value, and by adding appropriate new reaction(s).

Your code should use the `produce()` request to supply some items *after* requesting to consume them. Verify that this works!

Solution

```
val item = m[Int]
val consume = m[Unit]
val produce = m[Unit]
val n = 10
site(
  go { case item(x) + consume(_) ⇒ println(s"consumed $x") },
  go { case produce(_) ⇒ (1 to n).foreach(i ⇒ item(i)) }
)
consume() + consume() + consume()
produce()
```

Producer with given labels

Modify the previous program, allowing the produce request to specify a list of label values to be put on new items.

This should be modeled by the molecule `produce()` carrying a value of type `List[Int]` with an appropriate reaction.

Solution

```
val item = m[Int]
val consume = m[Unit]
val produce = m[List[Int]]
val n = 10
site(
  go { case item(x) + consume(_) ⇒ println(s"consumed $x") },
  go { case produce(ls) ⇒ ls.foreach(i ⇒ item(i)) }
)
consume() + consume() + consume()
produce(1 to n)
```

Take two

Modify the previous program, adding a request to consume 2 items.

This should be modeled by the molecule `consume2()` with `Unit` value.

Solution

```
val item = m[Int]
val consume = m[Unit]
val consume2 = m[Unit]
val produce = m[List[Int]]
val n = 10
site(
  go { case item(x) + consume(_) ⇒ println(s"consumed $x") },
  go { case produce(ls) ⇒ ls.foreach(i ⇒ item(i)) },
  go { case item(x) + item(y) + consume2(_) ⇒ println(s"consumed $x and $y") }
)
consume() + consume() + consume()
produce(1 to n)
consume2()
```

Batch consumer

Modify the previous program, adding request to consume `k` items, where `k` is an integer parameter.

This request should be modeled by a new molecule `consumeK()` carrying an integer value, and by adding appropriate new reaction(s).

Solution

```
val item = m[Int]
val consume = m[Unit]
val consumeK = m[Int]
val produce = m[List[Int]]
val n = 10
site(
  go { case item(x) + consume(_) ⇒ println(s"consumed $x") },
  go { case item(x) + consumeK(k) ⇒
    println(s"item $k consumed: $x")
    if (k > 0) consumeK(k - 1)
  },
  go { case produce(ls) ⇒ ls.foreach(i ⇒ item(i)) }
)
consume() + consume() + consume()
produce(1 to n)
consumeK(5)
```

Debugging

`Chymyst` has some debugging facilities to help the programmer verify that the chemistry works as intended.

Logging the contents of the soup

For debugging purposes, it is useful to see what molecules are currently present in the soup at a given reaction site (RS), waiting to react with other molecules. This is achieved by calling the `logSoup()` method on any of the molecule emitters. This method will return a string showing the molecules that are currently present in the soup at that RS. The output will also show the values carried by each molecule.

In our example, all three molecules `counter`, `incr`, and `decr` are declared as inputs at our RS, so we could use any of the emitters, say `decr`, to log the soup contents:

```
> println(decr.logSoup)
Site{counter + decr → ...; counter + incr → ...}
Molecules: counter(98)
```

The debug output contains two pieces of information:

- The RS which is being logged: `Site{counter + decr → ...; counter + incr → ...}` Note that the RS is identified by the reactions that are declared in it. The reactions are shown in a shorthand notation, which only mentions the input molecules.

- The list of molecules currently waiting in the soup at that RS, namely `Molecules: counter(98)`, showing that there is presently only one copy of the `counter` molecule, carrying the value `98`.

Also note that the debug output is limited to the molecules that are declared as *input* at that RS. We say that these molecules are **bound** to that RS. The RS will look at the presence or absence of these molecules when it decides which reactions to start.

Molecule names

A perceptive reader will ask at this point: How did the program know the names `counter`, `decr`, and `incr` when we called `logSoup()`? These are names of local variables we defined using `val counter = m[Int]` and so on. Ordinarily, Scala code does not have access to these names.

The magic is actually performed by the method `m`, which is a macro that looks up the name of the enclosing variable. The same effect can be achieved without macros at the cost of more boilerplate:

```
val counter = new M[Int]("counter")
// This is completely equivalent to `val counter = m[Int]`.
```

Molecule names are not checked for uniqueness and have no effect on program execution. Nevertheless, descriptive names of molecules are very useful for visualizing the reactions, as well as for debugging and logging. In this tutorial, we will always use macros to define molecules.

Common errors

Error: Emitting molecules with undefined chemistry

For each molecule, there must exist a single reaction site (RS) to which this molecule is bound — that is, the RS where this molecule is consumed as input molecule by some reactions. (See Reaction Sites for a more detailed discussion.)

It is an error to emit a molecule that is not yet defined as input molecule at any RS (i.e. not yet bound to any RS).

```
val x = m[Int]
x(100)
// java.lang.Exception: Molecule x is not bound to any reaction site
```

The same error will occur if the emitter call is attempted inside a reaction body, or if we call `logSoup()` on the molecule emitter.

The correct way of using `Chymyst` is first to define molecules, then to create a RS where these molecules are used as inputs for reactions, and only then to start emitting these molecules.

The method `isBound` can be used to determine at run time whether a molecule has been already bound to a RS:

```
val x = m[Int]

x.isBound // returns `false`

site( go { case x(2) ⇒ } )

x.isBound // returns `true`
```

Error: Redefining chemistry

Chemical laws are immutable in a `Chymyst` program. All reactions that consume a certain molecule must be declared in one reaction site. Once it is declared that a certain molecule starts certain reactions, users cannot add new reactions that consume that molecule.

For this reason, it is an error to write a reaction whose input molecule is already *used as input* at another reaction site.

```
val x = m[Int]
val a = m[Unit]
val b = m[Unit]

site( go { case x(n) + a(_) ⇒ println(s"got x($n) + a") } ) // OK, `x` is now bound to
 this RS.

site( go { case x(n) + b(_) ⇒ println(s"got x($n) + b") } )
// java.lang.Exception: Molecule x cannot be used as input in
// Site{b + x → ...} since it is already bound to Site{a + x → ...}
```

This program contradicts the intended meaning of reaction sites, namely that molecules arrive there to wait for their reaction partners. If we define two RSs as above and then emit `x(123)`, the molecule `x(123)` must go to the first RS if it is to react with `a()` and to the second RS if it is to react with `b()`. There is no single RS where `x()` could wait for both of its possible reaction partners, but `x()` cannot be present at two reaction sites at the same time!

What the programmer probably meant is simply that the molecule `x()` has two different reactions that consume it. Correct use of `Chymyst` requires that we put these two reactions together into *one* reaction site:

```
val x = m[Int]
val a = m[Unit]
val b = m[Unit]

site(
  go { case x(n) + a(_) ⇒ println(s"got x($n) + a") },
  go { case x(n) + b(_) ⇒ println(s"got x($n) + b") }
) // OK, this works.
```

More generally, all reactions that share any input molecules must be defined together in a single RS. Whenever a molecule is consumed by a reaction at some RS, we say that this molecule is **bound** to that RS. Any reactions consuming that molecule must be defined at the same RS.

However, reactions that use that molecule only as *output* may be declared in another RS. Here is an example where we define a reaction that computes a result and emits a molecule called `show`, which is bound to another RS:

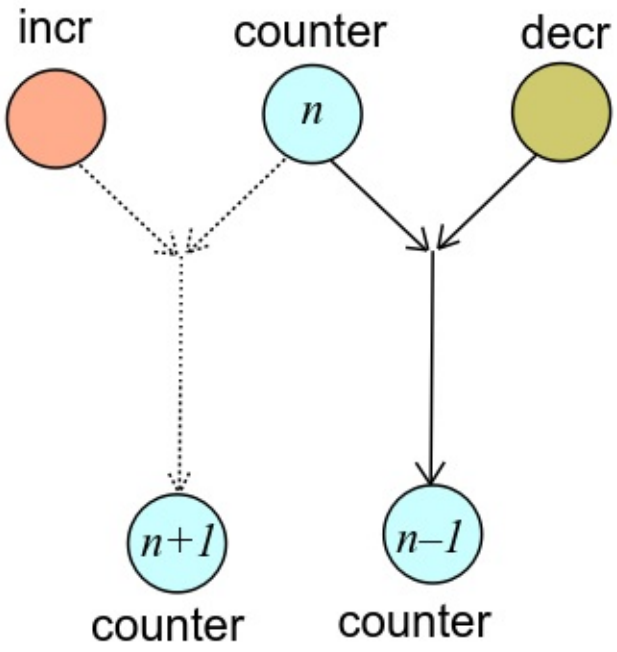
```
val show = m[Int]
// reaction site where the “show” molecule is an input molecule
site( go { case show(x) ⇒ println(s"got $x") })

val start = m[Unit]
// reaction site where the “show” molecule is an output molecule
// (but not an input molecule)
site(
  go { case start(_) ⇒ val res = compute(???); show(res) }
)
```

Order of reactions and indeterminism

When a reaction site has sufficiently many waiting molecules that several different reactions could start, the runtime engine will make a choice as to which reaction will actually be scheduled to start.

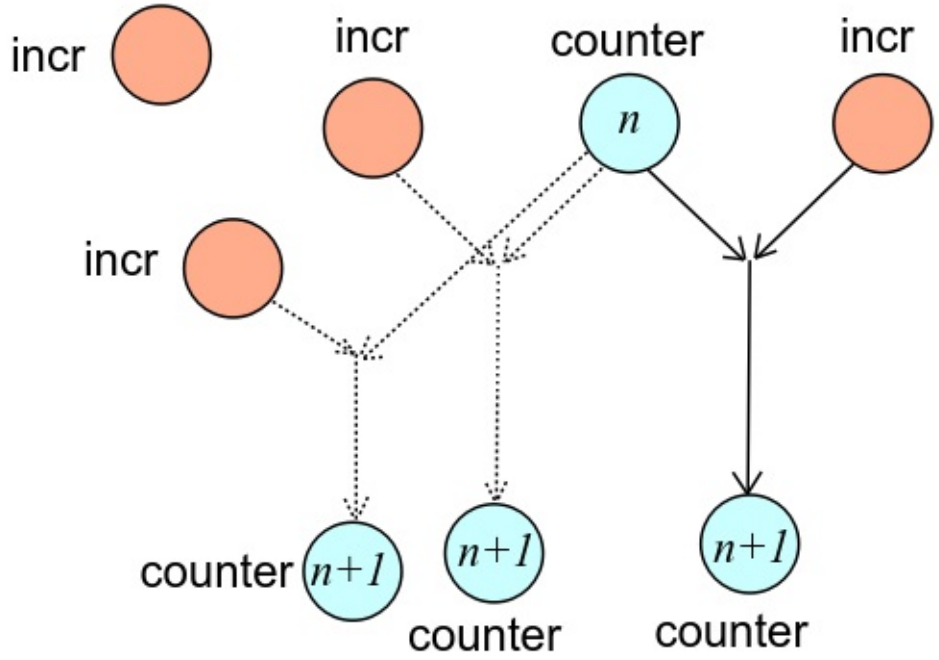
The next figure shows an example where the soup initially contains one copy of the `counter()` molecule, one copy of `incr()`, and one copy of `decr()`. The `counter` molecule could either react with the `incr()` molecule or with the `decr()` molecule. One of

these reactions (shown in solid lines) have been chosen to actually start, which leaves the second reaction (shown in dashed lines) without the input molecule `counter()`. Therefore, the second reaction cannot start now.

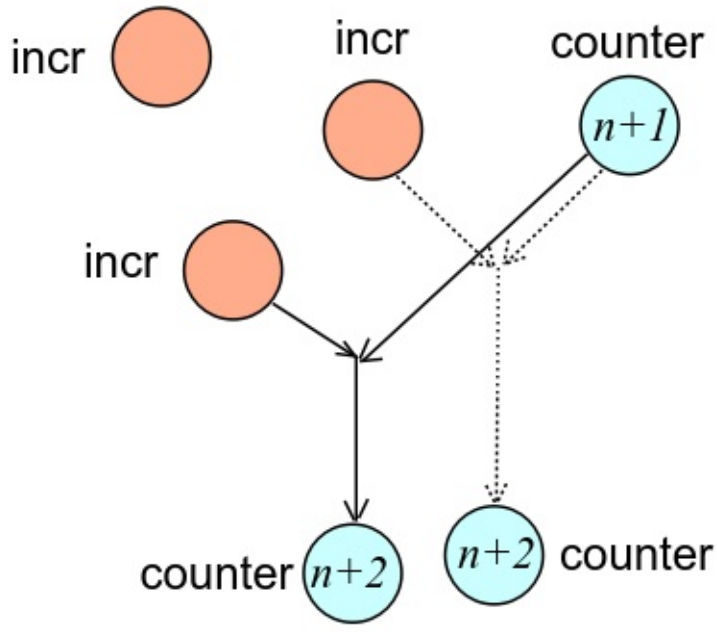


Similarly, when there are several copies of the *same* molecule that can be consumed by a reaction, the runtime engine will make a choice of which copy of the molecule to consume.

The next figure shows an example where the soup contains one copy of the `counter()` molecule and four copies of the `incr()` molecule. Each of the four `incr()` molecules can react with the one `counter()` molecule. The runtime engine is going to choose the molecules that will actually react. One reaction (shown in solid lines) will start, consuming the `counter()` and `incr()` molecules, while other possible reactions (some of them shown in dashed lines) will not start.



After this reaction, the soup will contain one copy of the `counter()` molecule (with the updated value) and the three remaining `incr()` molecules. At the next step, another one of the `incr()` molecules will be chosen to start a reaction, as shown in the next figure:



`Chymyst` does *not* randomize the order of input molecules or the order of reactions but makes an implementation-dependent choice, designed to optimize the performance of the reaction scheduler.

It is important to keep in mind that we *cannot* assign priorities to reactions or to input molecules. The chemical machine ignores the order in which reactions are listed in the `site(...)` call, as well as the order of molecules in the input list of each reaction. Within debugging messages, input molecules are printed in the alphabetical order of names, output molecules are printed in the order emitted, and reactions are printed in an unspecified but fixed order.

To summarize: When there are sufficiently many waiting molecules so that several different reactions could start, the order in which reactions will start is non-deterministic and unknown.

Concurrent programs usually expect a certain degree of indeterminism during execution. For example, the partial computations for a parallel map/reduce operation should be executed in the order that best fits the run-time conditions, rather than in any fixed order. The important requirement is that the final result should be deterministic.

If obtaining a deterministic final result in a particular application requires to maintain a fixed order for certain reactions, it is the programmer's task to design the chemistry in such a way that those reactions start in the desired order. This is always achievable by using appropriate auxiliary molecules and/or guard conditions.

In fact, a facility for assigning explicit priority ordering to molecules or reactions would be counterproductive! It will only give the programmer *an illusion of control* over the order of reactions, while actually introducing subtle non-deterministic behavior.

To illustrate this by example, suppose we would like to compute the sum of a bunch of numbers, where the numbers arrive concurrently at unknown times. In other words, we expect to receive many molecules `data(x)` with integer values `x`, and we need to compute and print the final sum value when no more `data(...)` molecules are present.

Here is an (incorrect) attempt to design the chemistry for this program:

```
// Non-working code!
val data = m[Int]
val sum = m[Int]
site (
// We really want the first reaction to be high priority...
  go { case data(x) + sum(y) ⇒ sum(x + y) },
// ...and run the second one only after all `data` molecules are gone.
  go { case sum(x) ⇒ println(s"sum = $x") }
)
data(5) + data(10) + data(150)
sum(0) // expect "sum = 165"
```

Our intention is to run only the first reaction and to ignore the second reaction as long as `data(...)` molecules are available in the soup. The chemical machine does not actually allow us to assign priorities to reactions. But, if we were able to assign a higher priority to the first reaction, what would be the result?

In reality, the `data(...)` molecules are going to be emitted at unpredictable times. For instance, they could be emitted by several other reactions that are running concurrently. Then it will sometimes happen that the `data(...)` molecules are emitted more slowly than we are consuming them at our reaction site. When that happens, there will be a brief interval of time when no `data(...)` molecules are present in the soup (although other reactions are perhaps about to emit some more of them). The chemical machine will then run the second reaction, consume the `sum(...)` molecule, and print the result, signalling (incorrectly) that the computation is finished. Perhaps this failure will *rarely* happen and will not be detected by unit tests, but at some point it is definitely going to happen in production code. This kind of indeterminism illustrates why concurrency is widely regarded as a hard programming problem.

`Chymyst` will actually reject our attempted program and print an error message before running anything, immediately after we define the reaction site:

```
val data = m[Int]
val sum = m[Int]
site (
  go { case data(x) + sum(y) ⇒ sum(x + y) },
  go { case sum(x) ⇒ println(s"sum = $x") }
)
```

```
Exception: In Site{data + sum → ...; sum → ...}: Unavoidable indeterminism: reaction {data + sum → } is shadowed by {sum → }
```

The error message means that the reaction `sum → ...` may prevent `data + sum → ...` from running, and that the programmer *has no control* over this indeterminism.

What we need here is to keep track of how many `data(...)` molecules we already consumed, and to print the final result only when we reach the total expected number of the `data(...)` molecules. Let us see how to implement this.

Since reactions do not hold any mutable state, the information about the remaining `data(...)` molecules has to be carried on the `sum(...)` molecule. So, we will define the `sum(...)` molecule with type `(Int, Int)`, where the second integer will be the number of `data(...)` molecules that remain to be consumed.

The reaction `data + sum` should proceed only when we know that some `data(...)` molecules are still remaining. Otherwise, the `sum(...)` molecule should start its own reaction and print the final result.

```
val data = m[Int]
val sum = m[(Int, Int)]
site (
  go { case data(x) + sum((y, remaining)) if remaining > 0 ⇒
    sum((x + y, remaining - 1))
  },
  go { case sum((x, 0)) ⇒ println(s"sum = $x") }
)
data(5) + data(10) + data(150) // emit three `data` molecules
sum((0, 3)) // "sum = 165" printed
```

Now the chemistry is correct; there is no need to assign priorities to reactions.

The chemical machine paradigm forces the programmer to design the chemistry in such a way that the order of running reactions is controlled by the data on the available molecules.

Another way of maintaining determinism when it is needed is to avoid writing reactions that might shadow each other's input molecules. Here is equivalent code with just one reaction:

```
val data = m[Int]
val sum = m[(Int, Int)]
site (
  go { case data(x) + sum((y, remaining)) ⇒
      val newSum = x + y
      if (remaining == 1)  println(s"sum = $newSum")
      else  sum((newSum, remaining - 1))
     }
)
data(5) + data(10) + data(150) // emit three `data` molecules
sum((0, 3)) // expect "sum = 165" printed
```

The drawback of this approach is that the chemistry became less declarative due to complicated branching code inside the reaction body. However, the program will run somewhat faster because no guard conditions need to be checked before scheduling a reaction, and thus the scheduler does not need to search for molecule values that satisfy the guard conditions.

If the run-time overhead of scheduling a reaction is insignificant compared with the computations inside reactions, the programmer may prefer to use a more declarative code style.

Summary so far

A chemical program consists of the following descriptions:

- the defined molecules, together with their value types;
- the reactions involving these molecules as inputs, together with reaction bodies;
- code that emits some molecules at the initial time.

These definitions comprise the “chemistry” of a concurrent program.

The user can define one or more reaction sites, each having one or more reactions. We imagine a reaction site to be a virtual place where molecules arrive and wait for other molecules, in order to start reactions with them. Each molecule has only one reaction site where that molecule can be consumed by reactions.

For this reason, all reactions that have a common *input* molecule must be declared at the same reaction site. Different reaction sites may not have any common input molecules.

By defining molecules, reaction sites, and the reactions at each site, we can specify an arbitrarily complicated system of interacting concurrent processes.

After defining the molecules and specifying the reactions, the code will typically emit some initial molecules into the soup. The chemical machine will then start running all the possible reactions, constantly keeping track of the molecules consumed by reactions or newly emitted into the soup.

Let us recapitulate the core ideas of the chemical paradigm of concurrency:

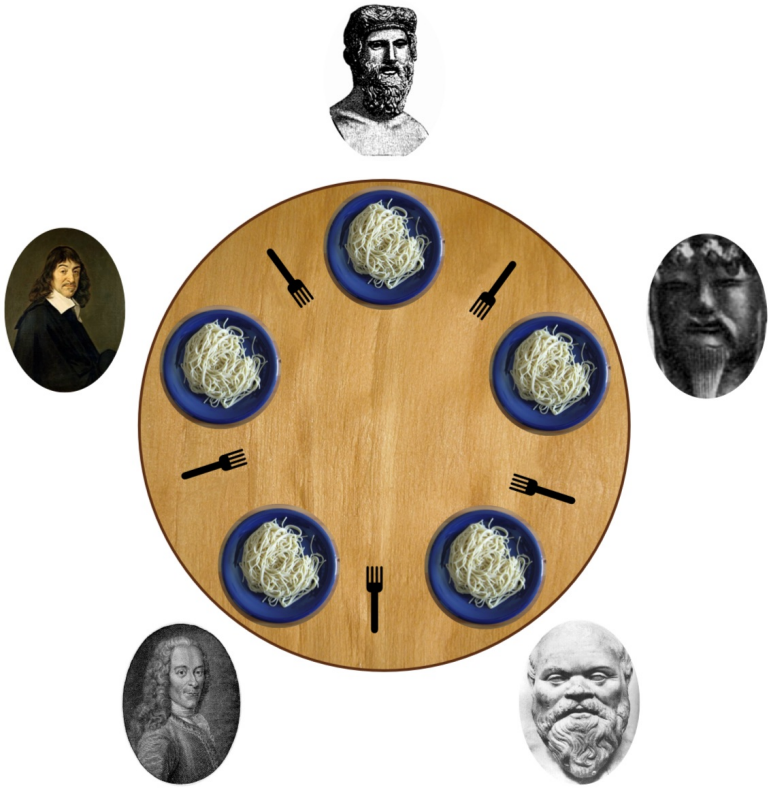
- In the chemical machine, there is no shared mutable state; all data is immutable and must be carried by some molecules.
- Each reaction specifies its input molecules, and in this way determines all the data necessary for computing the reaction body. The chemical machine will automatically make this data available to a reaction, since the reaction can start only when all its input molecules are present in the soup.
- A reaction body is a Scala expression that evaluates to `Any`. (The final result value of that expression is discarded.) The reaction body can perform arbitrary computations using the input molecule values or other values visible in the local scope.
- The reaction body will typically compute some new values and emit new molecules carrying these values. Emitting a molecule is a side effect of calling an emitter. Emitters can be called at any time — either within a reaction body or in any other code.
- Up to the side effect of emitting new molecules, the reaction body can be a pure function that only depends on the input data of the reaction. In this case, many copies of the reaction can be safely executed concurrently when many sets of input molecules are available. Also, the reaction can be safely and automatically restarted in the case of

a transient failure by simply emitting the input molecules again.

The chemical laws specify declaratively all the computations available for the data carried by the given molecules. Whenever multiple sets of data are available, the corresponding computations may be performed concurrently; this is decided automatically by the chemical machine.

Example: Declarative solution for “dining philosophers”

The “dining philosophers problem” is to run a simulation of five philosophers who take turns eating and thinking. Each philosopher needs two forks to start eating, and every pair of neighbor philosophers shares a fork.



The simplest solution of the “dining philosophers” problem in the chemical machine paradigm is achieved using a molecule for each fork and two molecules per philosopher: one representing a thinking philosopher and the other representing a hungry philosopher.

- Each of the five “thinking philosopher” molecules (`thinking1` , `thinking2` , ..., `thinking5`) starts a reaction in which the process is paused for a random time and then the “hungry philosopher” molecule is emitted.
- Each of the five “hungry philosopher” molecules (`hungry1` , ..., `hungry5`) needs to react with *two* neighbor “fork” molecules. The reaction process is paused for a random time,

and then the “thinking philosopher” molecule is emitted together with the two “fork” molecules previously consumed.

The complete code is shown here:

```
import io.chymyst.jc._

 /** Print message and wait for a random time interval. */
def wait(message: String): Unit = {
  println(message)
  Thread.sleep(scala.util.Random.nextInt(20))
}

val hungry1 = m[Unit]
val hungry2 = m[Unit]
val hungry3 = m[Unit]
val hungry4 = m[Unit]
val hungry5 = m[Unit]
val thinking1 = m[Unit]
val thinking2 = m[Unit]
val thinking3 = m[Unit]
val thinking4 = m[Unit]
val thinking5 = m[Unit]
val fork12 = m[Unit]
val fork23 = m[Unit]
val fork34 = m[Unit]
val fork45 = m[Unit]
val fork51 = m[Unit]

site (
  go { case thinking1(_) ⇒ wait("Socrates is thinking");  hungry1() },
  go { case thinking2(_) ⇒ wait("Confucius is thinking"); hungry2() },
  go { case thinking3(_) ⇒ wait("Plato is thinking");     hungry3() },
  go { case thinking4(_) ⇒ wait("Descartes is thinking"); hungry4() },
  go { case thinking5(_) ⇒ wait("Voltaire is thinking");  hungry5() },

  go { case hungry1(_) + fork12(_) + fork51(_) ⇒ wait("Socrates is eating");  thinking
1() + fork12() + fork51() },
  go { case hungry2(_) + fork23(_) + fork12(_) ⇒ wait("Confucius is eating"); thinking
2() + fork23() + fork12() },
  go { case hungry3(_) + fork34(_) + fork23(_) ⇒ wait("Plato is eating");     thinking
3() + fork34() + fork23() },
  go { case hungry4(_) + fork45(_) + fork34(_) ⇒ wait("Descartes is eating"); thinking
4() + fork45() + fork34() },
  go { case hungry5(_) + fork51(_) + fork45(_) ⇒ wait("Voltaire is eating");  thinking
5() + fork51() + fork45() }
)
// Emit molecules representing the initial state:
thinking1() + thinking2() + thinking3() + thinking4() + thinking5()
fork12() + fork23() + fork34() + fork45() + fork51()
// Now reactions will start and print messages to the console.
```

Note that a `hungry + fork + fork` reaction will consume a “hungry philosopher” molecule and two “fork” molecules, so these three molecules will not be present in the soup during the time interval taken by the “eating” process. Thus, neighbor philosophers will not be able to start eating until the two “fork” molecules are returned to the soup. Which philosophers start eating will be decided randomly, but there will never be a deadlock.

The example code shown above is *fully declarative*: it describes what the “dining philosophers” simulation must do but not how to do it, and the code is quite close to the English-language description of the problem.

The result of running this program is the output such as

```
Plato is thinking
Socrates is thinking
Voltaire is thinking
Descartes is thinking
Confucius is thinking
Plato is eating
Socrates is eating
Plato is thinking
Descartes is eating
Socrates is thinking
Voltaire is eating
Descartes is thinking
Confucius is eating
Voltaire is thinking
Plato is eating
Confucius is thinking
Socrates is eating
Socrates is thinking
Plato is thinking
Voltaire is eating
```

Chemical machine programming

The chemical machine can be programmed to perform arbitrary concurrent computations. However, it is not immediately obvious what molecules and reactions must be defined, say, to implement a concurrent buffered queue or a concurrent merge-sort algorithm. Another application would be a concurrent GUI interaction together with some jobs in the background.

Solving these problems via chemistry requires a certain paradigm shift. In order to build up our chemical intuition, let us go through some more examples, from simple to more complex.

Example: “Readers/Writers”

Suppose there is a single shared resource that can be accessed by a number of **Readers** and a number of **Writers**. The resource behavior is such that while a Writer is accessing the resource, no readers can have access; and vice versa. Additionally, either at most *one* Writer or at most *three* Readers should be able to access the resource concurrently.

To make our example concrete, we consider that the resource is “being accessed” when the given functions `readResource()` and `writeResource()` are being called.

The task is to create a chemical machine program that allows any number of concurrent Readers and Writers to call their respective functions but restricts the number of concurrent calls to at most three `readResource()` calls or at most one `writeResource()` call. The program should also prevent a `readResource()` and a `writeResource()` to be called at the same time.

Let us derive the solution in a systematic way by reasoning about this problem within the chemical machine paradigm.

We need to restrict code that calls certain functions. The only way a chemical machine can run any code is though running some *reactions*. Therefore, we need a reaction whose body calls `readResource()` and another reaction that calls `writeResource()`.

Thus, we need to define some input molecules that will start these reactions. Let us call these molecules `read()` and `write()` respectively:

```
val read = m[Unit]
val write = m[Unit]
site(
  go { case read(_) ⇒ readResource(); ??? },
  go { case write(_) ⇒ writeResource(); ??? }
)
```

Processes will emit `read()` or `write()` molecules when they need to access the resource as readers or as writers.

The reactions as written so far will always start whenever `read()` or `write()` are emitted. However, our task is to control when these reactions start. We need to *prevent* these reactions from starting when there are too many concurrent accesses.

In `Chymyst`, there are only two ways of preventing a reaction from starting:

- by withholding some of the required input molecules;
- by using a guard condition with a mutable variable, setting the value of that variable as required.

The second method requires complicated reasoning about the current values of mutable variables. Generally, shared mutable state is contrary to the spirit of functional programming, although it may be used in certain cases for performance optimization. Although Scala allows it, we will not use shared mutable state in `Chymyst`.

It remains to use the first method.

In order for us to be able to provide or withhold input molecules, the two reactions we just discussed need to have *another* input molecule. Let us call this additional molecule `access()` and revise the reactions accordingly:

```
val read = m[Unit]
val write = m[Unit]
val access = m[Unit]
site(
  go { case read(_) + access(_) ⇒ readResource(); ??? },
  go { case write(_) + access(_) ⇒ writeResource(); ??? }
)
access() // Emit at the beginning.
```

In this chemistry, a single `access()` molecule will allow one Reader or one Writer to proceed with work. However, after the work is done, the `access()` molecule will be gone, and no reactions will start. To remedy this, we need to emit `access()` at the end of both reactions:

```
val read = m[Unit]
val write = m[Unit]
val access = m[Unit]
site(
  go { case read(_) + access(_) ⇒ readResource(); access() },
  go { case write(_) + access(_) ⇒ writeResource(); access() }
)
access() // Emit at the beginning.
```

This implements Readers/Writers with *single* exclusive access for both. How can we enable 3 Readers to access the resource simultaneously?

We could emit 3 copies of `access()` at the beginning of the program run. However, this will also allow up to 3 Writers to access the resource. We would like to make it so that three Reader's accesses are equivalent to one Writer's access.

One way of doing this is simply to replace the single `access` molecule with three copies of `access` in Writer's reaction:

```
val read = m[Unit]
val write = m[Unit]
val access = m[Unit]
site(
  go { case read(_) + access(_) ⇒ readResource(); access() },
  go { case write(_) + access(_) + access() + access() ⇒
    writeResource(); access() + access() + access()
  }
)
access() + access() + access() // Emit three copies at the beginning.
```

This chemistry works as required! Any number of `read()` and `write()` molecules can be emitted, but the reactions will start only if sufficient `access()` molecules are present.

Generalizing Readers:Writers ratio to `n` : `1`

Our solution works but has a drawback: it cannot be generalized from 3 to `n` concurrent Readers, where `n` is a run-time parameter. This is so because our solution uses one `write()` molecule and 3 `access()` molecules as inputs for the second reaction. In order to generalize this to `n` concurrent Readers, we would need to write a reaction with `n + 1` input molecules. However, the input molecules for each reaction must be specified *at compile time*. So we cannot write a reaction with `n` input molecules, where `n` is a run-time parameter.

The only way to overcome this drawback is to count explicitly how many Readers have been granted access at the present time. The current count value must be updated every time we grant access to a Reader and every time a Reader finishes accessing the resource.

In the chemical machine, reactions are stateless, and the only way to keep and update a value is to put that value on some molecule, and to consume and emit this molecule in some reactions. Therefore, we need a molecule that carries the current reader count.

The easiest solution is to make `access()` carry an integer `k`, representing the current number of Readers that have been granted access. Initially we will have `k == 0`. We will allow a new `Writer` to have access only when `k == 0`, and a new `Reader` to have access only when `k < 3`.

As a first try, our reactions might look like this:

```
val read = m[Unit]
val write = m[Unit]
val access = m[Int]
val n = 3 // can be a run-time parameter
site(
  go { case read(_) + access(k) if k < n ⇒
    readResource(); access(k + 1)
  },
  go { case write(_) + access(0) ⇒ writeResource(); access(0) }
)
access(0) // Emit at the beginning.
```

The Writer reaction will now start only when `k == 0`, that is, when no Readers are currently reading the resource. This is exactly what we want.

The Reader reaction, however, does not work correctly for two reasons:

- It consumes the `access()` molecule for the entire duration of the `readResource()` call, preventing any other Readers from accessing the resource.
- After `readResource()` is finished, the reaction emits `access(k + 1)`, which incorrectly signals that now one more Reader is accessing the resource.

The first problem can be fixed by emitting `access(k + 1)` at the *beginning* of the reaction, before `readResource()` is called:

```
go { case read(_) + access(k) if k < n ⇒
  access(k + 1); readResource()
}
```

However, the second problem still remains. After `readResource()` is finished, we need to decrement the current value of `k` carried by `access(k)` at that time.

Since the values on molecules are immutable, the only way of doing this in the chemical machine is to *add another reaction* that will consume `access(k)` and emit `access(k - 1)`. To start this reaction, we obviously need another input molecule; let us call it `readerFinished()`:

```
go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
```

The new `readerFinished()` molecule will be emitted at the end of the Reader reaction.

The complete working code now looks like this:

```
val read = m[Unit]
val write = m[Unit]
val access = m[Int]
val readerFinished = m[Unit]
val n = 3 // can be a run-time parameter
site(
  go { case read(_) + access(k) if k < n ⇒
    access(k + 1)
    readResource()
    readerFinished()
  },
  go { case write(_) + access(0) ⇒ writeResource(); access(0) },
  go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
)
access(0) // Emit at the beginning.
```

Exercise

Modify this program to allow `m` simultaneous Readers and `n` simultaneous Writers to access the resource. One easy way of doing this would be to use negative integer values to count Writers who have been granted access.

Solution

```
val read = m[Unit]
val write = m[Unit]
val access = m[Int]
val readerFinished = m[Unit]
val writerFinished = m[Unit]
val nReader = 4
val nWriter = 3
site(
  go { case read(_) + access(k) if k >= 0 && k < nReader ⇒
    access(k + 1)
    readResource()
    readerFinished()
  },
  go { case write(_) + access(k) if -k >= 0 && -k < nWriter ⇒
    access(k - 1)
    writeResource()
    writerFinished()
  },
  go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
  go { case writerFinished(_) + access(k) ⇒ access(k + 1) }
)
access(0) // Emit at the beginning.
```

Passing values between reactions

The reactions written so far don't do much useful work besides synchronizing some function calls. How could we modify the program so that Readers and Writers exchange values with the resource?

Let us assume that the resource contains an integer value, so that `readResource()` returns an `Int` while `writeResource()` takes an `Int` argument.

Since we do not want to use any shared mutable state, the only way to pass values is to use molecules. So let us introduce a `readResult()` molecule with an `Int` value. This molecule will carry the value returned by `readResource()`.

Similarly, the `write()` molecule now needs to carry an `Int` value. When the `writeResource()` operation has been run (perhaps after some waiting for concurrent access), we will emit a new molecule `writeDone()`.

The revised code looks like this:

```
val read = m[Unit]
val readResult = m[Int]
val write = m[Int]
val writeDone = m[Unit]
val access = m[Int]
val readerFinished = m[Unit]
val n = 3 // can be a run-time parameter
site(
  go { case read(_) + access(k) if k < n ⇒
    access(k + 1)
    val x = readResource(); readResult(x)
    readerFinished()
  },
  go { case write(x) + access(0) ⇒ writeResource(x); access(0) + writeDone() },
  go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
)
access(0) // Emit at the beginning.
```

How would we use this program? A Reader is represented by a reaction that emits `read()`. However, the resulting value is carried by a new molecule `readResult(x)`. So the Reader client must implement *two* reactions: one will emit `read()` and the other will consume `readResult(x)` and continue the computation with the obtained value `x`.

```
// Reactions for a Reader: example
site(
  go { case startReader(_) ⇒ initializeReader(); read() },
  go { case readResult(x) ⇒ continueReader(x) }
)
```

We see a certain awkwardness in this implementation. Instead of writing a single reaction body for a Reader client, such as

```
// Single reaction for Reader: not working!
go { case startReader(_) ⇒
  val a = ???
  val b = ???
  val x = read() // This won't work since `read()` doesn't return `x`.
  continueReader(a, b, x)
}
```

we must break the code at the place of the `read()` call into two reactions:

- the first reaction will contain all code preceding the `read()` call,
- the second reaction will contain all code after the `read()` call, where the value `x` must be obtained by consuming the additional molecule `readResult(x)`.

Since the second reaction opens a new local scope, any local values (such as `a`, `b`) computed by the first reaction will be inaccessible. For this reason, we cannot rewrite the code shown above as two reactions like this:

```
// Two reactions for Reader: not working!
site(
  go { case startReader(_) ⇒
    val a = ???
    val b = ???
    read()
  },
  go { case readResult(x) ⇒
    continueReader(a, b, x)
// Where would `a` and `b` come from???
  }
)
```

The values of `a` and `b` would need to be passed to the second reaction somehow. Since the chemical machine does not support shared mutable state, we could pass `a` and `b` as additional values along with `read()` and `readResult()`. However, this is a cumbersome workaround that mixes unrelated concerns. The `read()` and `readResult()` molecules should be concerned only with the correct implementation of the concurrent access to the `readResource()` operation. These molecules should not be carrying any additional values that are used by other parts of the program and are completely unrelated to the `readResource()` operation.

This problem — breaking the scope into parts at risk of losing access to local variables — is sometimes called "stack ripping".

Similarly, the code for a Writer client must implement two reactions such as

```
// Reactions for a Writer: example
site(
  go { case startWriter(_) ⇒ initializeWriter(); val x: Int = ???; write(x) },
  go { case writeDone(_) ⇒ continueWriter() }
)
```

The scope of the Writer client must be split at the `write()` call.

We will see in the next chapter how to avoid stack ripping by using "blocking molecules". For now, we can use a trick that allows the second reaction to see the local variables of the first one. The trick is to define the second reaction *within the scope* of the first one:

```
// Two reactions for Reader: almost working but not quite
site(
  go { case startReader(_) ⇒
    val a = ???
    val b = ???
    read()
    // Define the second reaction site within the scope of the first one.
    site(
      go { case readResult(x) ⇒
        continueReader(a, b, x)
// Now `a` and `b` are obtained from the outer scope.
      }
    )
  }
)
```

There is still a minor problem with this code: the `readResult()` molecule has to be already defined when we write the reactions that consume `read()`, and yet we need to define a local new reaction that consumes `readResult()`. The solution is to pass the `readResult` emitter as a value on the `read()` molecule. In other words, the `read()` molecule will carry a value of type `M[Int]`, we will emit `read(readResult)`, and we will revise the reaction that consumes `read()` so that it emits `readResult()`.

The revised code looks like this:

```
// Working code for Readers/Writers access control.
val read = m[M[Int]]
val write = m[Int]
val writeDone = m[Unit]
val access = m[Int]
val readerFinished = m[Unit]
val n = 3 // can be a run-time parameter
site(
  go { case read(readResult) + access(k) if k < n ⇒
    access(k + 1)
    val x = readResource(); readResult(x)
    readerFinished()
  },
  go { case write(x) + access(0) ⇒ writeResource(x); access(0) + writeDone() },
  go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
)
access(0) // Emit at the beginning.
```

Note that emitting `readResult()` requires it to be already bound to a reaction site. So we must create the reaction site that consumes `readResult` before emitting `read(readResult)`.

Here is some skeleton code for Reader client reactions:

```
site(
  go { case startReader(_) ⇒
    val a = ???
    val b = ???
    val readResult = m[Int]
    // Define the second reaction site within the scope of the first one.
    site(
      go { case readResult(x) ⇒
        continueReader(a, b, x)
// Now `a` and `b` are obtained from the outer scope.
      }
    )
// Only now, after `readResult` is bound to its RS,
// we can safely emit `read()`.
    read(readResult)
  }
)
```

Now we can write the Reader and Writer reactions in a more natural style, despite stack ripping. The price is adding some boilerplate code and passing the `readResult()` molecule as the value carried by `read()`. As we will see in the next chapter, this boilerplate goes away if we use blocking molecules.

Molecules and reactions in local scopes

It is perfectly admissible to define new reaction sites (and/or new molecule emitters) within the scope of an existing reaction. Reactions defined within a local scope are treated no differently from any other reactions. In fact, new molecule emitters, new reactions, and new reaction sites are always defined within *some* local scope, and they can be defined within the scope of a function, a reaction, or even within the local scope of a value:

```
val a: M[Unit] = {
// some local emitters
  val c = m[Int]
  val a = m[Unit]
// local reaction site
  site(
    go { case c(x) + a(_) ⇒ println(x); c(x + 1) } // whatever
  )
  c(0) // Emit this just once.
  a // Return this emitter to the outside scope.
}
```

The chemistry implemented here is an asynchronous counter that prints its current value and increments it whenever `a()` is emitted.

The result of defining this chemistry within the scope of a value block is to declare a new molecule emitter `a` as a top-level value in the outer scope.

The emitters `c` and `a` with their chemistry are active but invisible outside the scope of their block. Only the emitter `a` is returned as the result value of the block. So, the emitter `a` is now accessible as the value `a` in the outer scope.

This trick gives us the ability to hide some emitters and to encapsulate their chemistry, making it safe to use by outside code. In this example, the correct function of the counter depends on having a *single* copy of `c()` in the soup. If the user were to emit (by mistake) any further copies of `c()`, the incrementing functionality would become unpredictable since the reaction `c + a → ...` could consume any of the available copies of `c()`, and the user has no control over the resulting indeterminism.

Encapsulating the chemistry in a block scope prevents this error from happening. Indeed, the inner scope emits exactly one copy of `c(0)`. In the outer scope, the code has access to the emitter `a` and can emit molecules `a()` at will, but cannot emit any new copies of `c()` because the emitter `c` is hidden within the inner scope. Thus it is guaranteed that the encapsulated reactions involving `c()` will run correctly.

Example: Concurrent map/reduce

Consider the problem of implementing a concurrent map/reduce operation. This operation first takes an array of type `Array[A]` and applies a function `f : A ⇒ B` to each element of the array. This yields an `Array[T]` of intermediate results. After that, a “reduce”-like operation `reduceB : (B, B) ⇒ B` is applied to that array, and the final result of type `B` is computed.

This can be implemented in sequential code like this:

```
val arr : Array[A] = ???
arr.map(f).reduce(reduceB)
```

Our task is to implement all these computations concurrently — both the application of `f` to each element of the array and the accumulation of the final result.

Let us assume that the `reduceB` operation is associative and commutative and has a zero element (i.e. that the type `B` is a commutative monoid). In that case, we may apply the `reduceB` operation to array elements in arbitrary order, which makes our task easier.

Implementing the map/reduce operation does not actually require the full power of concurrency: a bulk synchronous processing framework such as Hadoop or Spark will do the job. Our goal is to come up with a chemical approach to concurrent map/reduce for tutorial purposes.

Since we would like to apply the function `f` concurrently to values of type `A`, we need to put all these values on separate copies of some "carrier" molecule.

```
val carrier = m[A]
```

We will emit a copy of the `carrier` molecule for each element of the initial array:

```
val arr : Array[A] = ???
arr.foreach(i ⇒ carrier(i))
```

Since the molecule emitter inherits the function type `A ⇒ Unit`, we could equivalently write this as

```
val arr : Array[A] = ???
arr.foreach(carrier)
```

As we apply `f` to each element, we will carry the intermediate results on molecules of another sort:

```
val interm = m[T]
```

Therefore, we need a reaction of this shape:

```
go { case carrier(x) ⇒ val res = f(x); interm(res) }
```

Finally, we need to gather the intermediate results carried by `interm()` molecules. For this, we define the "accumulator" molecule `accum()` that will carry the final result accumulated by going over all the `interm()` molecules.

```
val accum = m[T]
```

When all intermediate results are collected, we would like to print the final result. At first we might write reactions for `accum` like this:

```
// Non-working code!
go { case accum(b) + interm(res) ⇒ accum( reduceB(b, res) ) },
go { case accum(b) ⇒ println(b) }
```

Our plan is to emit an `accum()` molecule, so that this reaction will repeatedly consume every `interm()` molecule until all the intermediate results are processed. When there are no more `interm()` molecules, we will print the final accumulated result.

However, there is a serious problem with this implementation: We will not actually find out when the work is finished! Our idea was that the processing will stop when there are no `interm()` molecules left. However, the `interm()` molecules are produced by previous reactions, which may take time. We do not know when each `interm()` molecule will be emitted: there may be prolonged periods of absence of any `interm()` molecules in the soup, while some reactions are still busy evaluating `f()`. The second reaction can start at any time — even when some `interm()` molecules are going to be emitted very soon. The runtime engine cannot know whether any reaction is going to eventually emit some more `interm()` molecules, and so the present program is unable to determine whether the entire map/reduce job is finished. The chemical machine will sometimes run the second reaction too early.

It is the programmer's responsibility to organize the chemistry such that the “end-of-job” situation can be detected. The simplest way of doing this is to *count* how many `interm()` molecules have been consumed.

Let us change the type of `accum()` to carry a tuple `(Int, B)`. The first element of the tuple will now represent a counter, which indicates how many intermediate results we have already processed. Reactions with `accum()` will increment the counter; the reaction with `fetch()` will proceed only if the counter is equal to the length of the array.

```
val accum = m[(Int, B)]

go { case accum((n, b)) + interm(res) ⇒
  accum( (n + 1, reduceB(b, res)) )
},
go { case accum((n, b)) if n == arr.length ⇒ println(b) }
```

What value should we emit with `accum()` initially? When the first `interm(res)` molecule arrives, we will need to call `reduceB(x, res)` with some value `x` of type `B`. Since we assume that `B` is a monoid, there must be a special value, say `bZero`, such that `reduceB(bZero, res) == res`. So `bZero` is the value we need to emit on the initial `accum()` molecule.

We can now emit all `carrier` molecules and a single `accum((0, bZero))` molecule. Because of the guard condition, the reaction with `println()` will not run until all intermediate results have been accumulated.

Here is the complete code for this example. We will apply the function `f(x) = x * x` to elements of an integer array and then compute the sum of the resulting array of squares.

```
import io.chymyst.jc._

object C1 extends App {

  // declare the "map" and the "reduce" functions
  def f(x: Int): Int = x * x
  def reduceB(acc: Int, x: Int): Int = acc + x

  val arr = 1 to 100

  // declare molecule types
  val carrier = m[Int]
  val interm = m[Int]
  val accum = m[(Int,Int)]

  // declare the reaction for the "map" step
  site(
    go { case carrier(x) ⇒ val res = f(x); interm(res) }
  )

  // The two reactions for the "reduce" step must be together since they both consume
`accum`.
  site(
      go { case accum((n, b)) + interm(res) ⇒ accum( (n + 1, reduceB(b, res)) ) },
      go { case accum((n, b)) if n == arr.length ⇒ println(b) }
  )

  // emit molecules
  accum((0, 0))
  arr.foreach(i ⇒ carrier(i))
 // prints "338350"
}
```

Achieving parallelism

The code in the previous subsection works correctly but has an important drawback: Since there is always at most one copy of the `accum()` molecule present, the reaction `accum + interm → ... reduceB() ...` cannot run concurrently with other instances of itself. In other words, at most one call to `reduceB()` will run at any given time! We would like to have some parallelism, so that multiple calls to `reduceB()` may run at once.

Since the `reduceB()` operation is commutative and associative, we may reduce intermediate results in any order. The easiest way of implementing this in the chemical machine would be to write a reaction such as

```
go { case interm(x) + interm(y) ⇒ interm(reduceB(x, y)) }
```

As `interm()` molecules are emitted, this reaction could run between any available pairs of `interm()` molecules. When running on a multi-core CPU, the chemical machine should be able to schedule many such reactions concurrently and optimize the CPU load.

This code, however, is not yet correct. When all `interm()` molecule pairs have reacted, the single `interm()` molecule will remain inert in the soup, since no other molecules can react with it. To fix this problem, we need a means of tracking progress and detecting when the entire computation is finished.

In our previous solution, we kept track of progress by using a counter `n` on the `accum()` molecule. Let us therefore add a counter also to the `interm()` molecule. The presence of an `interm((n, x))` molecule indicates that the partial reduce of a set of `n` numbers was already completed, with the result value `x`.

The reaction is rewritten like this:

```
site(
  go { case interm((n1, x1)) + interm((n2, x2)) ⇒
    interm((n1 + n2, reduceB(x1, x2)))
  },
  go { case interm((n, x)) if n == arr.length ⇒ println(x) }
)
```

This will work correctly if we initially emit `interm((1, x))`, indicating that the value `x` is a result of a partial reduce of a single-number set. Here is the complete sample code:

```
import io.chymyst.jc._

object C2 extends App {

  // declare the "map" and the "reduce" functions
  def f(x: Int): Int = x * x
  def reduceB(acc: Int, x: Int): Int = acc + x

  val arr = 1 to 100

  // declare molecule types
  val carrier = m[Int]
  val interm = m[(Int, Int)]

  // declare the reaction for the "map" step
  site(
    go { case carrier(x) ⇒ val res = f(x); interm((1, res)) }
  )

  // The two reactions for the "reduce" step must be together
  // since they both consume `accum`.
  site(
    go { case interm((n1, x1)) + interm((n2, x2)) ⇒
      val x = reduceB(x1, x2)
      val n = n1 + n2
      if (n == arr.length) println(x) else interm((n, x))
    }
  )

  // emit initial molecules
  arr.foreach(i ⇒ carrier(i))
 // prints 338350
}
```

Exercise

Modify the concurrent map/reduce program for the case when the `reduceB()` computation is itself asynchronous.

Assume that the computation of `reduceB` is defined as a reaction at another reaction site, with code such as

```
val inputsB = m[(Int, Int)]
val resultB = m[Int]

site(
  go { case inputsB((x, y)) ⇒
  val z = reduceB(x, y) // long computation
  resultB(z)
  }
)
```

Adapt the code in `object C2` from the previous section to use the molecules `inputsB()` and `resultB()`, instead of calling `reduceB()` directly.

Ordered map/reduce

Typically, the reduce operation is associative, but it may or may not be commutative. A simple example of an associative but non-commutative operation on integers is Budden's function,

```
def addBudden(x: Int, y: Int) = if (x % 2 == 0) x + y else x - y
```

If the `reduceB()` operation is non-commutative, we may not apply the reduce operation to just *any* pair of partial results. The map/reduce code in the previous subsection will select pairs in arbitrary order and will most likely fail to compute the correct final value for non-commutative reduce operations.

For instance, suppose we have an array `x1, x2, ..., x10` of intermediate results that we need to reduce with a non-commutative `reduceB()` operation. We may reduce `x4` with `x3` or with `x5`, but not with `x6` or any other element. Also, we need to reduce elements in the correct order, e.g. `reduceB(x3, x4)` but not `reduceB(x4, x3)`.

Once we have computed the new intermediate result `reduceB(x3, x4)`, we may reduce that with `x5`, with `x2`, or with the result of `reduceB(x1, x2)` or `reduceB(x5, x6, x7)` — but not with, say, `x6` or with `reduceB(x6, x7)`.

What we need to do is to restrict the `reduceB()` operation so that it runs only on *consecutive* partial results. How can we modify the chemistry to support these restrictions?

We need to assure that the reaction `interm + interm → interm` only consumes molecules that represent consecutive intermediate results, but does not start for any other pairs of input molecules.

The chemical machine has only two ways of preventing a reaction from starting:

1. by withholding some input molecules required for that reaction
2. by specifying guard conditions on input molecule values

If we were to use the first method, we would need to model all the allowed reactions with special auxiliary input molecules. We would have to define a new input molecule for each possible intermediate result: first, for each element of the initial array; then for each possible reduction between two consecutive elements; then for each possible reduction result between those, and so on. While this is certainly possible to implement, it would require is to define $O(n^2)$ different molecules and $O(n^3)$ different reactions, where n is the number of the initial `interm()` molecules before first `reduceB()` is called.
This means a very large number of possible molecules and reactions for the scheduler to choose from: we will need $O(n^3)$ operations just to define the chemistry for this code, which will then run only $O(n)$ reduce steps. The code will run unacceptably slowly if implemented in this way.

The solution with the second method is to add a guard condition to the reaction `interm + interm → interm`, so that it will only run between consecutive intermediate results. To identify such intermediate results, we need to put an ordering label on the `interm()` molecule values, which will allow us to write a reaction like this:

```
go { case interm((l1, x1)) + interm((l2, x2))
 if isConsecutive???(l1, l2) ⇒
  interm((???, reduceB(x1, x2)))
}
```

Note that `interm()` previously carried a counter value, which we need to keep track of the progress of the computation. The ordering label we denoted by `l1` and `l2` must be set in addition to that counter.

Let us decide that the value of the ordering label should be equal to the smallest index that has been reduced. Thus, the `interm()` molecule must carry a triple such as `interm((l, n, x))`, representing an intermediate result `x` that was computed after reducing `n` numbers. For example, `reduceB(reduceB(x5, x6), x7)` would be represented by `interm((5, 3, x))`.

With this representation, we can easily check whether two intermediate results are consecutive: the condition is `l1 + n1 == l2`. In this way, we combine the functions of the counter and the ordering label.

The “reduce” reaction can be now written as

```
go { case interm((l1, n1, x1)) + interm((l2, n2, x2))
  if l1 + n1 == l2 ⇒
  interm((l1, n1 + n2, reduceB(x1, x2)))
}
```

Other code remains unchanged, except for emitting the initial `interm()` molecules during the “map” step:

```
go { case carrier(i) ⇒ val res = f(i); interm((i, 1, res)) }
```

The performance of this code is significantly slower than that of the commutative map/reduce, because the chemical machine must go through many copies of the `interm()` molecule before selecting the ones that can react together. To improve performance, we can allow the two `interm()` molecules to react in any order:

```
go { case interm((l1, n1, x1)) + interm((l2, n2, x2))
  if l1 + n1 == l2 || l2 + n2 == l1 ⇒
   if (l1 + n1 == l2)
    interm((l1, n1 + n2, reduceB(x1, x2)))
   else
    interm((l2, n1 + n2, reduceB(x2, x1)))
}
```

This optimization is completely mechanical: it consists of permuting the order of repeated molecules before applying the guard condition. The chemical machine could perform this code transformation automatically for all such reactions. As of version 0.2.0, `Chymyst` does not implement this optimization.

Improving performance

The solution we derived in the previous subsection uses a single reaction with repeated input molecules and guard conditions. This type of reactions typically causes slow performance of the chemical machine. Contention on repeated input molecules with cross-molecule guard conditions will force the chemical machine to enumerate many possible combinations of input molecules before scheduling a new reaction. To improve the performance, we need to avoid such reactions.

For simplicity, we will now focus only on the “reduce” part of “map/reduce”. We will assume that the data consists of an array of initial values to which the `reduceB()` operation must be applied. The operation is assumed to be associative but non-commutative.

Consider the order in which the `reduceB` operation is to be applied to the elements of the initial array. In the previous solutions, we allowed *any* two consecutive values to be reduced at any time. This requires a complicated chemistry and, as a consequence, yields slow performance. Let us instead restrict the set of possible `reduceB()` operations: We will only permit the merging of elements 0 with 1, then 2 with 3, and so on; then we will repeat the same merging procedure recursively, effectively building a binary tree of `reduceB` operations.

To make this construction easier, let us begin with a hard-coded binary tree of reactions for the special case where we have exactly 8 intermediate results. All initial values need to be carried by molecules that we will denote by `a0()`, `a1()`, and so on. At the first step, we would like to permit merging `a0` with `a1`, `a2` with `a3`, `a4` with `a5`, and `a6` with `a7` (but no other pairs). After this merging, we expect to obtain four intermediate results: `a01`, `a23`, `a45`, and `a67`.

This is represented by the following chemistry:

```
site(
  go { case a0(x) + a1(y) ⇒ a01(reduceB(x,y)) },
  go { case a2(x) + a3(y) ⇒ a23(reduceB(x,y)) },
  go { case a4(x) + a5(y) ⇒ a45(reduceB(x,y)) },
  go { case a6(x) + a7(y) ⇒ a67(reduceB(x,y)) }
)
```

We will now permit merging of `a01` with `a23` and `a45` with `a67` (but no other `reduceB` operations). That will yield the two intermediate results, `a03` and `a47`:

```
site(
  go { case a01(x) + a23(y) ⇒ a03(reduceB(x,y)) },
  go { case a45(x) + a67(y) ⇒ a47(reduceB(x,y)) }
)
```

It remains to merge `a03` and `a47`, obtaining the final result `a07`:

```
site(
  go { case a03(x) + a67(y) ⇒ a07(reduceB(x,y)) }
)
```

Note that we could define all these reactions in separate reaction sites because there is no contention on the input molecules, and thus *all* `reduceB()` reactions can run concurrently. The performance of this reaction structure will be much better than that of the reactions with repeated molecules and guard conditions.

What remains is for us to be able to define this kind of reaction structure dynamically, at run time.

Note that all necessary reactions are almost identical and differ only in the molecules they consume and produce. We begin by defining an auxiliary function that creates one such reaction and new molecule emitters for it:

```
def reduceOne[T](res: M[T]): (M[T], M[T]) = {
  val a0 = m[T]
  val a1 = m[T]
  site( go { case a0(x) + a1(x) ⇒ res(reduceB(x,y)) } )
  (a0, a1)
}
```

The argument of this function is the result molecule emitter `res` that needs to be defined in advance.

We can now refactor our example 8-value chemistry by using this function. We have to start from the top result molecule `a07()` and descend towards the bottom:

```
val a07 = m[T]
val (a03, a47) = reduceOne(a07)

val (a01, a23) = reduceOne(a03)
val (a45, a67) = reduceOne(a47)

val (a0, a1) = reduceOne(a01)
val (a2, a3) = reduceOne(a23)
val (a4, a5) = reduceOne(a45)
val (a6, a7) = reduceOne(a67)

a0(...) + a1(...) + ... // emit all initial values on these molecules
```

Once we emit `a0()`, `a1()`, etc., the code will eventually emit `a07()` with the final result.

So far, this is entirely equivalent to hand-coded reactions for 8 initial values. It remains to transform the code to allow us to specify the number of initial values (8) as a run-time parameter. Recursion is the obvious solution here. Let us refactor the above code using an auxiliary recursive function that takes an array `arr` of initial values as argument. The auxiliary function will also emit the initial values once we reach the bottom level of the tree where the required emitter will be available as the parameter `res`:

```
def reduceAll[T](arr: Array[T], res: M[T]) =
  if (arr.length == 1) res(arr(0)) // Emit initial values.
  else  {
    val (arr0, arr1) = arr.splitAt(arr.length / 2)
    val (a0, a1) = reduceOne(res)
    reduceAll(arr0, a0)
    reduceAll(arr1, a1)
 }
```

What we have done is a simple refactoring of code in terms of a recursive function. This refactoring would be the same in any programming language and is not specific to chemical machine programming.

Let us now inline the call to `reduceOne()` and rewrite the code as a self-contained function:

```
def reduceAll[T](arr: Array[T], res: M[T]) =
  if (arr.length == 1) res(arr(0))
  else  {
    val (arr0, arr1) = arr.splitAt(arr.length / 2)
    val a0 = m[T]
    val a1 = m[T]

    site( go { case a0(x) + a1(y) ⇒ res(reduceB(x, y)) } )

    reduceAll(arr0, a0)
    reduceAll(arr1, a1)
 }
```

Note that the chemistry involving new molecules `a0()` and `a1()` is encapsulated within the scope of the function `reduceAll()`. The new molecules cannot be emitted outside that scope.

This solution works but has a defect: the function `reduceAll()` is not tail-recursive. To remedy this, we can refactor the body of the function `reduceAll()` into a *reaction*.

We declare `reduceAll` as a molecule emitter with value type `(Array[T], M[T])`. Instead of a recursive function, we obtain non-recursive code that can be thought of as a “chain reaction”:

```
val reduceAll = m[(Array[T], M[T])]
site(
 go { case reduceAll((arr, res)) ⇒
  if (arr.length == 1) res(arr(0))
  else  {
    val (arr0, arr1) = arr.splitAt(arr.length / 2)
    val a0 = m[T]
    val a1 = m[T]
    site( go { case a0(x) + a1(y) ⇒ res(reduceB(x, y)) } )
    reduceAll((arr0, a0)) + reduceAll((arr1, a1))
  }
 }
)
// start the computation:
val result = m[T]
val array: Array[T] = ... // create the initial array
reduceAll((array, result)) // start the computation
// The result() molecule will be emitted with the final result.
```

Nested reactions

What exactly happens when a new reaction is defined within the scope of an existing reaction? Each time the "parent" reaction `reduceAll() ⇒ ...` is run, a *new* pair of molecule emitters `a0` and `a1` is created. Then we call the `site()` function with a reaction that consumes the molecules `a0()` and `a1()`. The `site()` call will create a *new* reaction site and bind the new molecules `a0()` and `a1()` to that reaction site.

The new reaction site and the new molecule emitters `a0` and `a1` will be visible only within the scope of the parent reaction's body. In this way, the chemical machine works seamlessly with Scala's mechanism of local scopes. It guarantees that no other code could disturb the intended functionality of the reactions encapsulated within the scope of `reduceAll()`.

Since the `reduceAll()` reaction emits its own input molecules until the array is fully split into individual elements, it will run many times to define the new reactions we previously denoted by `r01`, `r23`, `r45`, `r67`, `r03`, `r47`, and `r07`. Thus, emitting the initial molecule `reduceAll((arr, res))` will create a tree-like structure of chain reactions at run time, reproducing the tree-like computation structure that we previously hand-coded.

Exercise:* concurrent ternary search

The task is to search through a sorted array `arr: Array[Int]` for a given value `x`. The result must be of type `Option[Int]`, indicating the index at which `x` is present in the array, or `None` if not found.

The well-known binary search will divide the array in two parts, determine the part where `x` might be present, and run the search recursively in that part. This algorithm cannot be parallelized. However, if we divide the array into *three* parts, we can check concurrently whether `x` is below the first or the second division point. This is the “ternary search” algorithm.

Implement the chemistry that performs the ternary search, as a recursive reaction that generates the necessary tree of computations.

Example: Concurrent merge-sort

As we have just seen in the previous section, chemical programs can implement recursion: A molecule can start a reaction whose reaction body defines further reactions and emits the same molecule, which will start another copy of the same reaction, etc. One can visualize this situation as a “chain reaction” (of course, proper precautions are taken so that the computations eventually terminate).

Since each reaction body will have a fresh local scope, the chain reaction will define *chemically new molecules and new reactions* each time. This will create a recursive configuration of reactions, such as a linked list or a tree.

We will now figure out how to use chain reactions for implementing the well-known “merge sort” algorithm in `Chymyst`.

The initial data will be an array of type `T`, and we will therefore need a molecule to carry that array. We will also need another molecule, `sorted()`, to carry the sorted result.

```
val mergesort = m[Array[T]]
val sorted = m[Array[T]]
```

The main idea of the merge-sort algorithm is to split the array in half, sort each half recursively, and then merge the two sorted halves into the resulting array.

```
site(
  go { case mergesort(arr) ⇒
    if (arr.length == 1)
      sorted(arr) // all done, trivially
    else {
      val (part1, part2) = arr.splitAt(arr.length / 2)
      // emit recursively
      mergesort(part1) + mergesort(part2)
      ???
    }
  }
)
```

We still need to merge pairs of sorted arrays. Let us assume that an array-merging function `arrayMerge(arr1, arr2)` is already implemented. We could then envision a reaction like this:

```
go { case sorted1(arr1) + sorted2(arr2) ⇒
  sorted( arrayMerge(arr1, arr2) )
}
```

Actually, we need to return the *upper-level* `sorted` molecule after merging the results carried by the lower-level `sorted1` and `sorted2` molecules. In order to achieve this, we can define the merging reaction within the scope of the `mergesort` reaction:

```
site(
  go { case mergesort(arr) ⇒
    if (arr.length == 1)
      sorted(arr) // all done, trivially
    else {
      val (part1, part2) = arr.splitAt(arr.length / 2)
      // define lower-level "sorted" molecules
      val sorted1 = m[Array[T]]
      val sorted2 = m[Array[T]]
      site(
        go { case sorted1(arr1) + sorted2(arr2) ⇒
         sorted( arrayMerge(arr1, arr2) ) // all done, merged
        }
      )
      // emit recursively
      mergesort(part1) + mergesort(part2)
    }
  }
)
```

This is still not quite right; we need to arrange the reactions such that the `sorted1()` and `sorted2()` molecules are emitted by the lower-level recursive emissions of `mergesort`. The way to achieve this is to pass the emitters for the upper-level `sorted` molecules on values

carried by the `mergesort()` molecule. Let us make the `mergesort()` molecule carry both the array and the upper-level `sorted` emitter. We will then be able to pass the lower-level `sorted` emitters to the recursive calls of `mergesort()`.

```
val mergesort = new M[((Array[T], M[Array[T]]))]

site(
  go {
    case mergesort((arr, sorted)) ⇒
      if (arr.length <= 1)
        sorted(arr) // all done, trivially
      else {
        val (part1, part2) = arr.splitAt(arr.length/2)
        // `sorted1` and `sorted2` will be the sorted results from lower level
        val sorted1 = new M[Array[T]]
        val sorted2 = new M[Array[T]]
        site(
          go { case sorted1(arr1) + sorted2(arr2) ⇒
            sorted(arrayMerge(arr1, arr2)) // all done, merged
          }
        )
        // emit lower-level mergesort
        mergesort(part1, sorted1) + mergesort(part2, sorted2)
      }
  }
)
// sort our array at top level, assuming `finalResult: M[Array[T]]`
mergesort((array, finalResult))
```

The complete working example of the concurrent merge-sort is in the file `MergesortSpec.scala`.

Blocking vs. non-blocking molecules

Motivation for the blocking molecule feature

So far, we have used molecules whose emission was a non-blocking call. Emitting a molecule `a()`, for example by calling its emitter as `a(123)`, immediately returns `Unit` but performs a concurrent side effect, adding a new copy of the molecule `a()` to the soup. Such molecules are called **non-blocking**.

When a reaction emits a non-blocking molecule, that molecule could later cause another reaction to start and compute some result. However, the emitting reaction continues to run concurrently and therefore has no direct access to the result computed by another reaction. It is sometimes convenient to be able to wait until the other reaction starts and computes the result, and then to obtain the result value in the first reaction.

Here is some skeleton code that illustrates the problem.

```
val a = m[Int]
val result = m[Int]

site(
  go { case a(x) ⇒
    val y = f(x) // some computation
    result(y)
  }
)
a(123)
// Waiting for `result()` to be emitted.
// We would like to obtain `y` here
// and continue computations with it.
```

This feature is realized in the chemical machine with help of **blocking molecules**.

How blocking molecules work

Just like non-blocking molecules, a blocking molecule carries a value and can be emitted into the soup. However, there are two major differences:

- emitting a blocking molecule is a function call that returns a **reply value** of a fixed type, rather than `Unit`;
- emitting a blocking molecule will block the emitting reaction or thread, until some

reaction consumes the blocking molecule and sends the reply value.

Here is an example of declaring a blocking molecule:

```
val f = b[Unit, Int]
```

The blocking molecule `f` carries a value of type `Unit`; the reply value is of type `Int`. We can choose these types at will.

This is how we can emit the blocking molecule `f` and wait for the reply:

```
val x: Int = f() // blocking emitter
```

The chemical machine emits a blocking molecule in a special way:

1. The emission call, such as `f(x)`, will add a new copy of the molecule `f(x)` to the soup, — this is so with any molecule.
2. However, the original emitting process (which can be a reaction body or any other code) will be blocked at least until *some other* reaction starts running and consumes the emitted copy of `f(x)`.
3. Once such a reaction starts, that reaction's body will **send a reply value** to the original process that emitted `f()`. Sending a reply value is accomplished by calling a special **reply emitter**, which is only available within the scope of reactions that consume blocking molecules.
4. The original process becomes unblocked right after it receives the reply value. To the original process, the reply value appears to be the value returned by the function call `f(x)`. Within the reaction that sends a reply, the reply call is asynchronous (it returns immediately, without waiting for the unblocked process).

Here is an example showing the `Chymyst` syntax for emitting a reply. Suppose we have a reaction that consumes a non-blocking molecule `c()` with an integer value and the blocking molecule `f()` defined above. We would like to use the blocking molecule in order to fetch the integer value that `c()` carries. The typical code for this kind of reaction looks like this:

```
val f = b[Unit, Int]
val c = m[Int]

site( go { case c(n) + f(_ , rpl) ⇒ rpl(n) } )

c(123) // emit a molecule `c` with value 123

val x = f() // now x = 123
```

Let us walk through the execution of this code step by step.

The blocking molecule `f()` is defined with two type parameters `[Unit, Int]`. The `Unit` type is the value it carries; the `Int` type is the reply value it receives.

After defining the chemical law `c + f → ...`, we first emit an instance of `c(123)`. By itself, this does not start any reactions since the chemical law states `c + f → ...`, and we don't yet have any copies of `f()` in the soup. So `c(123)` will remain in the soup for now, waiting at the reaction site.

Next, we call `f()`, which emits an instance of `f()` into the soup. Now the soup has both `c` and `f`. According to the semantics of blocking molecules, the calling process is blocked until a reaction consuming `f()` can start.

At this point, we have such a reaction: this is `c + f → ...`. This reaction is actually ready to start since both its inputs, `c()` and `f()`, are now present. Nevertheless, the start of that reaction is concurrent with the process that calls `f()`, and may occur somewhat later than the call to `f()`, depending on the CPU load. So, the call to `f()` will be blocked for some (hopefully short) time.

Once the reaction `c + f → ...` starts, it will consume both `c(123)` and `f()` and receive the value `n = 123` from the input molecule `c(123)`. The value `rpl` is defined within the scope of the reaction as the *second* pattern variable in the input molecule pattern `f(_, rpl)`. The value `rpl` is the reply emitter for the blocking molecule `f()`, and the reaction calls it as `rpl(n)` with `n = 123`. This call sends the reply value `123` back to the calling process. After that, the calling process will get unblocked and receive `123` as the return value of the function call `f()`.

From the point of view of the reaction `c + f → ...`, sending a reply is a non-blocking (i.e. a very fast) function call. So, reply emitters are similar to non-blocking molecules.

After replying, the reaction `c + f → ...` will continue evaluating whatever code follows the reply emitter call. The newly unblocked process that received the reply value will continue to run *concurrently* with the reaction that replied.

We see that blocking molecules work at once as synchronizing barriers) and as channels of communication between processes.

The syntax for the reaction,

```
go { case c(n) + f(_, rpl) ⇒ rpl(n) }
```

makes it appear as if the molecule `f()` carries *two* values — its ordinary `Unit` value and the reply emitter value `rpl`. However, `f` is emitted with the syntax `f()` — just as any other molecule with `Unit` value. The `rpl` emitter appears *only* in the pattern-matching expression for `f` inside a reaction. We call `rpl` the **reply emitter** because the call to `rpl()` has a concurrent side-effect similar to emitting a non-blocking molecule.

Note that the name `rpl` is not a keyword but an arbitrary name of a pattern variable; it could have been `x` or `reply_to_f` or whatever else.

Blocking molecule emitters are values of type `B[T, R]`, while non-blocking molecule emitters have type `M[T]`. Here `T` is the type of value that the molecule carries, and `R` (for blocking molecules) is the type of the reply value. The reply emitter is of special type `ReplyEmitter[T, R]`.

In general, the pattern-matching expression for a blocking molecule `g` of type `B[T, R]` has the form

```
{ case ... + g(v, r) + ... ⇒ ... }
```

The pattern variable `v` will match a value of type `T`. The pattern variable `r` will match the reply emitter.

The reply emitter must be matched with a simple pattern variable. For clarity, we will usually name this pattern variable `reply` or `r`. It is an error to use a non-variable pattern for the reply emitter.

Example: Benchmarking the asynchronous counter

To illustrate the usage of non-blocking and blocking molecules, let us consider the task of *benchmarking* the asynchronous counter we have previously defined. The plan is to initialize the counter to a large value *N*, then to emit *N* decrement molecules, and finally wait until the counter reaches the value 0. We will use a blocking molecule to wait until this happens and thus to determine the time elapsed during the countdown.

Let us now extend the previous reaction site to implement this new functionality. The simplest solution is to define a blocking molecule `fetch()`, which will react with the counter molecule only when the counter reaches zero. Since the `fetch()` molecule does not need to pass any data, we will define it as type `Unit` with a `Unit` reply.

```
val fetch = b[Unit, Unit]
```

We can implement this reaction by using a guard in the `case` clause:

```
go { case fetch(_, reply) + counter(n) if n == 0  ⇒ reply() }
```

For more clarity, we can also use Scala's pattern matching facility to implement the same reaction like this:

```
go { case counter(0) + fetch(_, reply)  ⇒ reply() }
```

Here is the complete code:

```
import io.chymyst.jc._


object C3 extends App {

  // declare molecule types
  val fetch = b[Unit, Unit]
  val counter = m[Int]
  val decr = m[Unit]

  // declare reactions
  site(
    go { case counter(0) + fetch(_, reply)  ⇒ reply() },
    go { case counter(n) + decr(_) ⇒ counter(n - 1) }
  )

  // emit molecules

  val N = 10000
  val initTime = System.currentTimeMillis()
  counter(N)
  (1 to N).foreach( _ ⇒ decr() )
  fetch()
  val elapsed = System.currentTimeMillis() - initTime
  println(s"Elapsed: $elapsed ms")
}
```

Some remarks:

- We declare both reactions in one reaction site because these two reactions contend on the common input molecule `counter`.
- Blocking molecules are like functions except that they will block until their reactions can start. If the relevant reaction never starts, — for instance, because some input molecules are missing, — a blocking molecule will block forever. The runtime engine cannot prevent this situation, because it cannot determine whether the currently missing

input molecules might become available in the future.

- The correct functionality of a chemical program may depend on the order in which blocking molecules are emitted. If `f()` and `g()` are blocking molecules, the effect of emitting `f(); g()` is not the same as that of `g(); f()` because `f(); g()` will not emit `g()` until a reply to `f` is received. With non-blocking molecules, the order of emission is irrelevant since the emission calls are asynchronous. So `c(); d()` has the same effect as `d(); c()` if `c` and `d` are non-blocking emitters.
- If several reactions can consume a blocking molecule, one of these reactions will be selected arbitrarily.
- Blocking molecule names are printed with the suffix `"/B"` in the debugging output.
- Molecules with unit values can be emitted simply by calling `decr()` and `fetch()` without arguments, but they still require a pattern variable when used in the `case` construction. For this reason, we need to write `decr(_)` and `fetch(_, reply)` in the match patterns.

Example: Readers/Writers with blocking molecules

Previously, we implemented the Readers/Writers problem using non-blocking molecules.

The final code looked like this:

```
// Code for Readers/Writers access control, with non-blocking molecules.
val read = m[M[Int]]
val write = m[(Int, M[Unit])]
val access = m[Int]
val readerFinished = m[Unit]
val n = 3 // can be a run-time parameter
site(
  go { case read(readResult) + access(k) if k < n ⇒
    access(k + 1)
    val x = readResource(); readResult(x)
    readerFinished()
  },
  go { case write((x, writeDone)) + access(0) ⇒ writeResource(x); access(0) + writeDon
e() },
  go { case readerFinished(_) + access(k) ⇒ access(k - 1) }
)
access(0) // Emit at the beginning.
```

Let us see what changes are necessary if we want to use blocking molecules, and what advantages that brings.

We would like to change the code so that `read()` and `write()` are blocking molecules. With that change, reactions that emit `read()` or `write()` are going to be blocked until access is granted. This will make the Readers/Writers functionality easier to use. (The cost is that some threads will become blocked.)

When we change the types of the `read()` and `write()` molecules to blocking, we will have to emit replies in reactions that consume `read()` and `write()`. We can also omit `readResult()` since the `read()` molecule will now receive a reply value. Other than that, reactions remain essentially unchanged:

```
// Code for Readers/Writers access control, with blocking molecules.
val read = b[Unit, Int]
val write = b[Int, Unit]
val access = m[Int]
val finished = m[Unit]
val n = 3 // can be a run-time parameter
site(
  go { case read(_, readReply) + access(k) if k < n ⇒
    access(k + 1)
    val x = readResource(); readReply(x)
    finished()
  },
  go { case write(x, writeReply) + access(0) ⇒ writeResource(x); writeReply(); access(0
) },
  go { case finished(_) + access(k) ⇒ access(k - 1) }
)
access(0) // Emit at the beginning.
```

Note the similarity between this blocking version and the previous non-blocking version. The reply molecules play the role of the auxiliary molecules `readResult()` and `writeDone()`. Instead of passing these auxiliary molecules on values of `read()` and `write()`, we simply declare the molecules `read()` and `write()` as blocking molecules and get the `writeReply` and `readReply` emitters automatically.

Also, the Reader and Writer client programming is now significantly streamlined compared to the non-blocking version: we do not need the boilerplate previously used to mitigate the "stack ripping" problem.

```
// Code for Reader client reactions, with blocking molecules.
site(
  go { case startReader(_) ⇒
    val a = ???
    val b = ???
    val x = read() // This is blocked until we get access to the resource.
    continueReader(a, b, x)
  }
)
```

A side benefit is that emitting `read()` and `write()` will get unblocked only *after* the `readResource()` and `writeResources()` operations are complete.

The price for this convenience is that the Reader thread will remain blocked until access is granted. The non-blocking version of the code never blocks any threads, which improves parallelism.

The unblocking transformation

By comparing two versions of the Readers/Writers code, we notice that there is a correspondence between blocking and non-blocking code styles. A reaction that emits a blocking molecule is equivalent to two reactions, using a new auxiliary non-blocking molecule defined in the scope of the first reaction. The reply emitter is replaced by an ordinary non-blocking molecule emitter whose corresponding molecule triggers a reaction that computes the required continuation.

The following two code snippets illustrate the correspondence.

The initial code snippet:

```
// Reaction that emits a blocking molecule.
val blockingMol = b[T, R]
go { case ... ⇒
  /* start of reaction body 1, defines `a`, `b`, ... */
  val t: T = ???
  val x: R = blockingMol(t)
  /* continuation of reaction body 1, can use `x`, `a`, `b`, `t`, ... */
}

// Reaction that consumes the blocking molecule.
go { case blockingMol(t, reply) + ... ⇒
  /* start of reaction body 2, uses `t`, defines `x` */
  val x: R = ???
  reply(x) // reply emitter
  /* rest of reaction body 2 */
}
```

The same code after the unblocking transformation:

```
// Reaction that emits a non-blocking molecule.
val nonBlockingMol = m[(T, M[R])] // this replaces `blockingMol`
go { case ... ⇒
  /* start of reaction body 1, defines `a`, `b`, ... */
  val t: T = ???
  val auxReply = m[T] // auxiliary reply emitter
  site(
    go { case auxReply(x) ⇒
    /* continuation of reaction body 1, can use `x`, `a`, `b`, `t`, ... */
    }
  )
  nonBlockingMol((t, auxReply)) // this replaces the call `blockingMol(t)`
}

// Reaction that consumes the non-blocking molecule.
go { case nonBlockingMol((t, reply)) + ... ⇒
  /* start of reaction body 2, uses `t`, defines `x` */
  val x: R = ???
  reply(x) // ordinary, non-blocking molecule emitter
  /* rest of reaction body 2 */
}
```

This example is the simplest case where the blocking molecule is emitted in the middle of a simple list of declarations within the reaction body. In order to transform this code into non-blocking code, the reaction body is split at the point where the blocking molecule is emitted. The rest of the reaction body is then moved into a nested auxiliary reaction that consumes `auxReply()`.

This code transformation can be seen as an optimization: When we translate blocking code into non-blocking code, we improve efficiency of the CPU usage because fewer threads will be waiting. For this reason, it is desirable to perform this **unblocking transformation** when possible.

If the blocking molecule is emitted inside an `if-else` or a `match-case` block, the unblocking transformation will need more work. Sometimes it will be necessary to introduce multiple auxiliary reply molecules and multiple nested reactions, corresponding to all the possible continuations of the reaction body. Also, the transformation may need to duplicate some parts of the reaction body's Scala code.

As an example, consider this reaction:

```
val a = m[Int]
val f = b[Unit, Int]
val g = b[Unit, Int]
val report = m[Int]

site(go { case a(x) ⇒
  val result = if (x > 0) {
    1 + f()
  } else {
    2 + g()
  }
  // use `result` here
  report(result)
})
```

The important detail is that the reply values of `f()` and `g()` are used for intermediate computations inside the `if-else` block. In this example, the computations are simply adding 1 or 2 to the reply values, but in general there can be arbitrary further computations within each of the two clauses. Because of this, the unblocking transformation needs two auxiliary reply molecules, and the Scala code following the `if-else` block needs to be duplicated:

```
val a = m[Int]
val f = m[M[Int]]
val g = m[M[Int]]
val report = m[Int]
val reply1 = m[Int]
val reply2 = m[Int]

site(go { case a(x) ⇒
  site(
    go { case reply1(r) ⇒
      val result = 1 + r
      // use `result` here
      report(result)
    },
    go { case reply2(r) ⇒
      val result = 2 + r
     // use `result` here
      report(result) }
  )
  if (x > 0) f(reply1) else g(reply2)
})
```

The duplicated code could be refactored into an auxiliary function or, as shown in the code snippet above, into an emitted molecule `report()`, but the duplication of the `report()` call is unavoidable.

Similarly, the unblocking transformation becomes more involved when several blocking molecules are emitted sequentially within a reaction body:

```
val a = m[Int]
val f = b[Unit, Int]
val g = b[Unit, Int]
val report = m[Int]

site(go { case a(x) ⇒
  val result = f() + g()
  // use `result` here
  report(result)
})
```

The unblocking transformation must be performed first for the `f()` and then for the `g()` call:

```
val a = m[Int]
val f = b[Unit, Int]
val g = b[Unit, Int]
val report = m[Int]

site(go { case a(x) ⇒
  val result = f() + g()
  // use `result` here
  report(result)
})
```

There are some cases where the unblocking transformation is problematic. One such case is a blocking molecule emitted inside a loop, or more generally, within a function scope:

```
val f = b[Unit, Int]
def callF(): Int = {
  val x = f()
  val y = f()
  x + y
}
```

The unblocking transformation requires us to replace `val x = f()` by an auxiliary reaction such as

```
go { case reply(res) ⇒
  val x = res
  // continuation goes here!
}
```

However, the function `callF()` could be invoked elsewhere in the application code, and we cannot insert a fixed continuation code into the auxiliary reaction shown above. In such cases, it is impossible to perform the unblocking transformation automatically. The programmer would need to redesign the program manually, - for example, replacing the function `callF()` by a molecule emitter `callF` with the corresponding reactions.

In order to ensure the possibility of the unblocking transformation for reaction bodies, `Chymyst` currently prohibits emitting blocking molecules in code contexts that are not guaranteed to be evaluated exactly once:

```
val c = m[Unit]
val f = b[Int, Int]
go { case c(_) ⇒ (1 to 10).map(i ⇒ f(i + 1)) }
```

`Error:(245, 8) reaction body must not emit blocking molecules inside function blocks (f(?)) go { case c(_) ⇒ (1 to 10).map(i ⇒ f(i + 1)) }`

In a future version of `Chymyst`, the unblocking transformation may be performed by macros as an automatic optimization.

The unblocking transformation and continuations

The unblocking transformation is closely related to programming in the continuation-passing style (CPS).

Let us convert to CPS a code snippet we used to illustrate the unblocking transformation.

The initial code snippet:

```
// Reaction that emits a blocking molecule.
val blockingMol = b[T, R]
go { case ... ⇒
  /* start of reaction body 1, defines `a`, `b`, ... */
  val t: T = ???
  val x: R = blockingMol(t)
  /* continuation of reaction body 1, can use `x`, `a`, `b`, `t`, ... */
}

// Reaction that consumes the blocking molecule.
go { case blockingMol(t, reply) + ... ⇒
  /* start of reaction body 2, uses `t`, defines `x` */
  val x: R = ???
  reply(x)
  /* rest of reaction body 2 */
}
```

The same code after CPS transformation:

```
// Reaction that emits a non-blocking molecule.
// The molecule now carries a continuation.
val nonBlockingMol = m[(T, R ⇒ Unit)]
go { case ... ⇒
  /* start of reaction body 1, defines `a`, `b`, ... */
  val t: T = ???
  val cont = { x : R ⇒
    /* continuation of reaction body 1, can use `x`, `a`, `b`, `t`, ... */
    }
  nonBlockingMol((t, cont))
}

// Reaction that consumes the non-blocking molecule.
go { case nonBlockingMol((t, cont)) + ... ⇒
  /* start of reaction body 2, uses `t`, defines `x` */
  val x: R = ???
  cont(x) // invoke continuation
  /* rest of reaction body 2 */
}
```

An important difference is that the continuation is invoked in the same process as reaction body 2. In other words, the continuation is not executing concurrently with the rest of reaction body 2. This loss of concurrency does not happen when using the unblocking transformation or with the original code that uses blocking molecules.

Molecules and reactions in local scopes

Since molecules and reactions are local values, they are lexically scoped within the block where they are defined. If we define molecules and reactions in the scope of an auxiliary function, or in the scope of a reaction body, these newly defined molecules and reactions will be encapsulated and protected from outside access.

We already saw one use of this feature for the unblocking transformation, where we defined a new reaction site nested within the scope of a reaction. To further illustrate this feature of the chemical paradigm, let us implement a function that encapsulates an “asynchronous counter” and initializes it with a given value.

Our first implementation of the asynchronous counter has a drawback: The molecule `counter(n)` must be emitted by the user and remains globally visible. If the user emits two copies of `counter()` with different values, the `counter + decr` and `counter + fetch` reactions will work unreliably, choosing between the two copies of `counter()` non-deterministically. In order to guarantee reliable functionality, we would like to emit exactly one copy of `counter()` and then prevent the user from emitting any further copies of that molecule.

A solution is to define `counter` and its reactions within a function that returns the `decr` and `fetch` emitters to the outside scope. The `counter` emitter *will not* be returned to the outside scope, and so the user will not be able to emit extra copies of that molecule.

```
def makeCounter(initCount: Int): (M[Unit], B[Unit, Int]) = {
  val counter = m[Int]
  val decr = m[Unit]
  val fetch = m[Unit, Int]

  site(
    go { counter(n) + fetch(_, r) ⇒ counter(n); r(n)},
    go { counter(n) + decr(_) ⇒ counter(n - 1) }
  )
  // emit exactly one copy of `counter`
  counter(initCount)

  // return only these two emitters to the outside scope
  (decr, fetch)
}
```

The function scope creates the emitter for the `counter()` molecule and emits a single copy of that molecule. Users from other scopes cannot emit another copy of `counter()` since the emitter is not visible outside the function scope. In this way, it is guaranteed that one and only one copy of `counter()` will be emitted into the soup.

Nevertheless, the users obtain the emitters `decr` and `fetch` from the function. So the users can emit these molecules and start the corresponding reactions.

The function `makeCounter()` can be called like this:

```
val (d, f) = makeCounter(10000)
d() + d() + d() // emit 3 decrement molecules
val x = f() // fetch the current value
```

Also note that each invocation of `makeCounter()` will create new, fresh emitters `counter`, `decr`, and `fetch`, because each invocation will create a fresh local scope and a new reaction site. In this way, the user can create as many independent counters as desired. Molecules defined in the scope of a certain invocation of `makeCounter()` will be *chemically different* from molecules defined in other invocations, since they will be bound to the particular reaction site defined during the same invocation of `makeCounter()`.

This example shows how we can encapsulate some molecules and yet use their reactions. A function scope can define local reactions with several input molecules, emit some of these molecules initially, and return some (but not all) molecule emitters to the outer scope.

Example: implementing “First Result”

The “First Result” operation solves the following problem. Suppose we have two blocking molecules `f()` and `g()` that return a reply value. We would like to emit both `f()` and `g()` together and wait until a reply value is received from whichever molecule unblocks sooner. If the other molecule gets a reply value later, we will just ignore that value.

The result of this non-deterministic operation is the value of type `T` obtained from one of the molecules `f` and `g`, depending on which molecule got its reply first.

Let us now implement this operation in `Chymyst`. We will derive the required chemistry by reasoning about the behavior of molecules.

The task is to define a blocking molecule emitter `firstReply` that will unblock when `f` or `g` unblocks, whichever happens first. Let us assume for simplicity that both `f()` and `g()` return values of type `Int`.

It is clear that we need to emit both `f()` and `g()` concurrently. But we cannot do this from one reaction, since `f()` will block and prevent us from emitting `g()`. Therefore we need two different reactions, one emitting `f()` and another emitting `g()`.

These two reactions need some input molecules. These input molecules cannot be `f` and `g` since these two molecules are given to us, their chemistry is already fixed, and so we cannot add new reactions that consume them. Therefore, we need at least one new molecule that will be consumed to start these two reactions. However, if we declare the two reactions as `c → f` and `c → g` and then emit two copies of `c()`, we are not guaranteed that both reactions will start. It is possible that two copies of the first reaction or two copies of the second reaction are started instead. In other words, there will be an *unavoidable indeterminism* in our chemistry.

`Chymyst` will in fact detect this problem and generate an error:

```
val c = m[Unit]
val f = b[Unit, Int]
val g = b[Unit, Int]
site(
    go { case c(_) ⇒ val x = f(); ??? },
    go { case c(_) ⇒ val x = g(); ??? }
)
```

```
java.lang.Exception: In Site{c → ...; c → ...}: Unavoidable indeterminism: reaction {c → } is shadowed by {c → }
```

So, we need to define two *different* molecules (say, `c` and `d`) as inputs for these two reactions.

```
val c = m[Unit]
val d = m[Unit]
val f = b[Unit, Int]
val g = b[Unit, Int]
site(
    go { case c(_) ⇒ val x = f(); ??? },
    go { case d(_) ⇒ val x = g(); ??? }
)
c() + d()
```

After we have emitted both `c()` and `d()`, both reactions will start concurrently. When one of them finishes and gets a reply value `x`, we would like to use that value for replying to our new blocking molecule `firstResult`.

Can we reply to `firstResult` in the same reactions? This would require us to make `firstResult` an input molecule in each of these reactions. However, we will have only one copy of `firstResult` emitted. Having `firstResult` as input will therefore prevent both reactions from proceeding concurrently.

Therefore, there must be some *other* reaction that consumes `firstResult` and replies to it. This reaction must have the form

```
go { case firstResult(_, reply) + ??? ⇒ reply(x) }
```

In this reaction, we somehow need to obtain the value `x` that we will reply with. So we need a new auxiliary molecule, say `done(x)`, that will carry `x` on itself. The reaction with `firstResult` will then have the form

```
go { case firstResult(_, reply) + done(x) ⇒ reply(x) }
```

Now it is clear that the value `x` should be emitted on `done(x)` in both of the `c → ...` and `d → ...` reactions. The complete program looks like this:

```
val firstResult = b[Unit, Int]
val c = m[Unit]
val d = m[Unit]
val f = b[Unit, Int]
val g = b[Unit, Int]
val done = m[Int]

site(
  go { case c(_) ⇒ val x = f(); done(x) },
  go { case d(_) ⇒ val x = g(); done(x) }
)
site(
  go { case firstResult(_, r) + done(x) ⇒ r(x) }
)

c() + d()
val result = firstResult()
```

Encapsulating chemistry in a function

The code as written works but is not encapsulated — in this code, we define new molecules and new chemistry at top level. To remedy this, we can create a function that will return the `firstResult` emitter, given the emitters `f` and `g`.

The idea is to define new chemistry in the function's *local scope* and return a new molecule emitter. Let us make this function generic by using a type parameter `T` for the result value of the given blocking molecules `f` and `g`. The code then looks like this:

```
def makeFirstResult[T](f: B[Unit, T], g: B[Unit, T]): B[Unit, T] = {
    // The same code as above, but now in a local scope.
    val firstResult = b[Unit, T]
    val c = m[Unit]
    val d = m[Unit]
    val f = b[Unit, T]
    val g = b[Unit, T]
    val done = m[T]

    site(
      go { case c(_) ⇒ val x = f(); done(x) },
      go { case d(_) ⇒ val x = g(); done(x) }
    )
    site(
      go { case firstResult(_, r) + done(x) ⇒ r(x) }
    )

    c() + d()

    // Return only the `firstResult` emitter:
    firstResult
}
```

The main advantage of this code is to encapsulate all the auxiliary molecule emitters within the local scope of the method `makeFirstResult()`. Only the `firstResult` emitter is returned; this is the only emitter that users of `makeFirstResult()` need. The other emitters (`c`, `d`, and `done`) are invisible to the users because they are local variables in the scope of `makeFirstResult`. Now the users of `makeFirstResult()` cannot inadvertently break the chemistry by emitting some further copies of `c`, `d`, or `done`. This is the hallmark of a successful encapsulation.

Example: Parallel Or

We will now consider a task called “Parallel Or”; this is somewhat similar to the “First Result” task considered above.

The goal is to create a new blocking molecule `parallelOr` from two given blocking molecules `f` and `g` that return a reply value of `Boolean` type. We would like to emit both `f` and `g` together and wait until a reply value is received. The `parallelOr` should reply with `true` as soon as one of the blocking molecules `f` and `g` returns `true`. If both `f` and `g` return `false` then `parallelOr` will also return `false`. If both molecules `f` and `g` block (or one of them returns `false` while the other blocks) then `parallelOr` will block as well.

We will now implement this operation in `Chymyst`. Our task is to define the necessary molecules and reactions that will simulate the desired behavior.

Let us recall the logic we used when reasoning about the "First Result" problem. We will again need two non-blocking molecules `c` and `d` that will emit `f` and `g` concurrently. We begin by writing the two reactions that consume `c` and `d`:

```
val c = m[Unit]
val d = m[Unit]
val f = b[Unit, Boolean]
val g = b[Unit, Boolean]

site(
  go { case c(_) ⇒ val x = f(); ??? },
  go { case d(_) ⇒ val y = g(); ??? }
)
c() + d()
```

Now, the Boolean values `x` and `y` could appear in any order (or not at all). We need to implement the logic of receiving these values and computing the final result value. This final result must be carried by some molecule, say `result()`. We should keep track of whether we already received both `x` and `y`, or just one of them.

When we receive a `true` value, we are done. So, we only need to keep track of the number of `false` values received. Therefore, let us define `result()` with integer value that shows how many intermediate `false` results we already received.

The next step is to communicate the values of `x` and `y` to the reaction that updates the `result()`'s value. For this, we can use a non-blocking molecule `done()` and write reactions like this:

```
site(
  go { case c(_) ⇒ val x = f(); done(x) },
  go { case d(_) ⇒ val y = g(); done(y) },
  go { case result(n) + done(x) ⇒ result(n + 1) }
)
c() + d() + result(0)
```

The effect of this chemistry is that `result()`'s value will get incremented whenever one of `f` or `g` returns. This is not yet what we need, but we are getting closer.

What remains is to implement the required logic: When `result` receives a `true` value, it should return `true` as a final answer, regardless of how many answers it received. When `result` receives a `false` value twice, it should return `false` as a final answer. Otherwise, there is no final answer yet.

We are required to deliver the final answer as a reply to the `parallelOr()` molecule. It is clear that `parallelOr` cannot be reacting with the `result()` molecule — this would prevent the `result + done → result` reaction from running. Therefore, `parallelOr` needs to react with *another* auxiliary molecule, say `finalResult(). There is only one way of defining this kind of reaction,

```
go { case parallelOr(_, r) + finalResult(x) ⇒ r(x) }
```

Now it is clear that the problem will be solved if we emit `finalResult` only when we actually have the final answer. This can be done from the `result + done → result` reaction, which we modify as follows:

```
go { case result(n) + done(x) ⇒
        if (x == true) finalResult(true)
        else if (n == 1) finalResult(false)
        else result(n + 1)
   }
```

To make the chemistry clearer, we may rewrite this reaction as three reactions with conditional pattern matching:

```
site(
  go { case result(1) + done(false) ⇒ finalResult(false) },
  go { case result(0) + done(false) ⇒ result(1) },
  go { case result(_) + done(true)  ⇒ finalResult(true) }
)
```

Here is the complete code for `parallelOr`, where we have separated the reactions into three independent reaction sites.

```
val c = m[Unit]
val d = m[Unit]
val done = m[Boolean]
val result = m[Int]
val finalResult = m[Boolean]
val f = b[Unit, Boolean]
val g = b[Unit, Boolean]
val parallelOr = b[Unit, Boolean]

site(
  go { case parallelOr(_, r) + finalResult(x) ⇒ r(x) }
)
site(
  go { case result(1) + done(false) ⇒ finalResult(false) },
  go { case result(0) + done(false) ⇒ result(1) },
  go { case result(_) + done(true)  ⇒ finalResult(true) }
)
site(
  go { case c(_) ⇒ val x = f(); done(x) },
  go { case d(_) ⇒ val y = g(); done(y) }
)

c() + d() + result(0)
```

Exercises

1. Encapsulate the `parallelOr` operation into a function.
2. Implement `parallelOr` for *three* blocking emitters (say, `f`, `g`, `h`). The `parallelOr()` call should return `true` whenever one of these emitters returns `true`; it should return `false` if *all* of them return `false`; and it should block otherwise.

Constraints on blocking molecules

No blocking at the end of reaction body

A reaction may not emit a blocking molecule as the last expression it computes.

```
val f = b[Unit, Int]
site( go { case ... ⇒ ...; f() } ) // the last expression is f()
// compile-time error: "Blocking molecules must not
// be emitted last in a reaction"
```

This code is considered to be a programmer's design error because blocking just before the end of the reaction and throwing away the received reply value is most likely useless. The blocking molecule `f()` should probably be replaced by a non-blocking molecule. If, for

some reason, the blocking molecule is required as the last expression, writing `f(); ()` will make the emitter call `f()` acceptable to `Chymyst` since it is no longer the last expression computed by the reaction.

One reply for each blocking molecule

Each blocking molecule must receive one (and only one) reply. It is an error if a reaction consumes a blocking molecule but does not reply. It is also an error to reply again after a reply was made.

Errors of this type are caught at compile time:

```
val f = b[Unit, Int]
val c = m[Int]
site( go { case f(_, r) + c(n) ⇒ c(n + 1) } ) // forgot to reply!
// compile-time error: "blocking input molecules should receive a reply
// but no unconditional reply found"

site( go { case f(_, r) + c(n) ⇒ c(n + 1); r(n); r(n) } ) // replied twice!
// compile-time error: "blocking input molecules should receive one reply
// but possibly multiple replies found"
```

The reply could depend on a run-time condition, which is impossible to evaluate at compile time. In this case, a reaction must use an `if` expression that calls the reply emitter in *each* branch. It is an error if a reply is only emitted in one of the `if` branches:

```
val f = b[Unit, Int]
val c = m[Int]
site( go { case f(_, r) + c(n) ⇒ c(n + 1); if (n != 0) r(n) } )
// compile-time error: "blocking input molecules should receive a reply
// but no unconditional reply found"
```

A correct reaction could look like this:

```
val f = b[Unit, Int]
val c = m[Int]
site(
  go { case f(_, r) + c(n) ⇒
    // reply is always sent, regardless of the value of `n`
    c(n + 1); if (n != 0) r(n) else r(0)
  }
)
```

It is a compile-time error to use a reply emitter inside a loop, inside any function block, or, more generally, in any **non-linear** context. (A context is "linear" if it is guaranteed to get evaluated exactly once.)

It is also an error to create an alias for a reply emitter or to pass it as an argument to functions.

All these restrictions are in place to ensure statically (at compile time) that a reply emitter will be called exactly once for each blocking molecule consumed by a reaction.

```
val f = b[Unit, Int]
site( go { case f(_, r) ⇒ val x = r; q(x) } )
// compile-time error: "Reaction body must not use reply emitters inside function bloc
ks"

site( go { case f(_, r) ⇒ try { throw ...; r(1) } catch {...} } )
// compile-time error: "Reaction body must not use reply emitters inside function bloc
ks"

site( go { case f(_, r) ⇒ if (r(1)) ... } ) // OK, the condition for `if` is evaluated
 exactly once
```

Avoid throwing exceptions in reaction body

A reaction's body could throw an exception before emitting a reply. In this case, compile-time analysis will not show that there is a problem because it is not possible to detect at compile time whether a portion of Scala code throws an exception. Nevertheless, the chemical machine will recognize at run time that the reaction body has finished without sending a reply to one or more blocking molecules. The chemical machine will then log an error message, specifying the molecules that are still waiting for replies.

Here is an example of code that emits `f()` and waits for reply, while a reaction consuming `f()` can throw an exception before replying:

```
val f = b[Unit, Int]
val c = m[Int]
site( go { case f(_, r) + c(n) ⇒ c(n + 1); if (n == 0) throw new Exception("Bad value
of n!"); r(n) } )
c(0)
f()
```

Running this code will print an error message to the console:

```
In Site{c + f/B → ...}: Reaction {c + f/B → ...} finished without replying
to f/B. Reported error: Bad value of n!
```

In general, a reaction can consume one or more blocking molecules and fail to reply to some of them due to an exception. Thus, there can be one or more blocked threads still waiting for replies when this kind of situation occurs. The runtime engine cannot know whether it is desirable to unblock these threads.

In our example, the exception will be thrown only when the reaction starts and the condition `n == 0` is evaluated to `true`. It may not be easy to design unit tests for this condition.

Another difficulty is that exceptions thrown in concurrent threads will be invisible in calling threads except for error messages on the console.

For these reasons, it is not easy to catch errors of this type, either at compile time or at run time.

To avoid these problems, it is advisable to design reactions in such a way that each reply is guaranteed to be sent exactly once, and that no exceptions can be thrown before sending the reply. If a condition needs to be checked before sending a reply, it should be a simple condition that is guaranteed not to throw an exception.

Example: map/reduce with blocking wait

In the previous chapter, we have seen the following code for the ordered map/reduce problem:

```
val reduceAll = m[(Array[T], M[T])]
site(
 go { case reduceAll((arr, res)) ⇒
  if (arr.length == 1) res(arr(0))
  else  {
    val (arr0, arr1) = arr.splitAt(arr.length / 2)
    val a0 = m[T]
    val a1 = m[T]
    site( go { case a0(x) + a1(y) ⇒ res(reduceB(x, y)) } )
    reduceAll((arr0, a0)) + reduceAll((arr1, a1))
  }
 }
)
// start the computation:
val result = m[T]
val array: Array[T] = ... // create the initial array
reduceAll((array, result)) // start the computation
// The result() molecule will be emitted with the final result.
```

We will now rewrite this code so that it waits until the entire computation is finished.

Presently, the molecule `result()` is emitted as the indication that the computation is done. We can easily introduce a blocking molecule `waitResult()` with a reaction that requires the `result()` molecule as another input:

```
val waitResult = B[Unit, T]

go { case waitResult(_, r) + result(x) ⇒ r(x) }
```

If we now emit `waitResult()`, it will block until its reaction can start, which will happen only when the map/reduce job is done. At that time, the final result of the computation, `x`, will be sent via the reply emitter `r()` to the process that emitted `waitResult()`.

We can now encapsulate the code as a (blocking) function call:

```
def doReduce[T](array: Array[T], reduceB: (T, T) ⇒ T): T = {
  val result = m[T]
  val waitResult = B[Unit, T]

  site( go { case waitResult(_, r) + result(x) ⇒ r(x) } )

  val reduceAll = m[(Array[T], M[T])]

  site(
   go { case reduceAll((arr, res)) ⇒
    if (arr.length == 1) res(arr(0))
    else  {
      val (arr0, arr1) = arr.splitAt(arr.length / 2)
      val a0 = m[T]
      val a1 = m[T]
      site( go { case a0(x) + a1(y) ⇒ res(reduceB(x, y)) } )
      reduceAll((arr0, a0)) + reduceAll((arr1, a1))
    }
   }
  )
  reduceAll((array, result)) // start the computation
  waitResult() // wait until finished and return the value.
}
```

Molecules and emitters, in depth

Molecule names

The names of molecules in `Chymyst` have no effect on any concurrent computations. For instance, the runtime engine will not check that a molecule's name is not empty, or that the names of different molecules are different. Molecule names are used only for debugging: they are printed when logging reactions and reaction sites.

There are two ways of assigning a name to a molecule:

- specify a name explicitly, by using a class constructor
- use the macros `m` and `b`

Here is an example of defining emitters using explicit class constructors and molecule names:

```
val counter = new M[Int]("counter")
val fetch = new B[Int, Int]("fetch")
```

This code is *completely equivalent* to the shorter code written using macros:

```
val counter = m[Int]
val fetch = b[Int, Int]
```

These macros read the names `"counter"` and `"fetch"` from the surrounding code. This functionality is intended as a syntactic convenience.

Each molecule emitter has a `toString` method, which returns the molecule's name. For blocking molecules, the molecule's name is followed by `"/B"`.

```
val x = new M[Int]("counter")
val y = new B[Unit, Int]("fetch")

x.toString // returns “counter”
y.toString // returns “fetch/B”
```

Molecules vs. molecule emitters

Molecules are emitted into the chemical soup using the syntax such as `c(123)`. Here, `c` is a value we define using a construction such as

```
val c = m[Int]
```

Any molecule emitted in the soup must carry a value. For example, the molecule `c()` must carry an integer value.

So the value `c` we just defined is not a molecule in the soup. The value `c` is a **molecule emitter**, — a function that, when called, will emit molecules of chemical sort `c` into the soup. More precisely, an emitter call such as `c(123)` returns `Unit` and performs a *side effect* that consists of emitting the molecule of sort `c` with value `123` into the soup.

The syntax `c(x)` is used in two different ways:

- in the left-hand side of a reaction, it is a pattern matching construction that matches on values of input molecules
- in the reaction body, it is an emitter call

In both cases, `c(x)` can be visualized as a molecule with value `x` that exists in the soup.

The chemistry-resembling syntax such as `c(x) + d(y)` is also used in two different ways:

- in the left-hand side of a reaction, it is a pattern matching construction that matches on values of several input molecules at once
- in the reaction body, it is syntactic sugar for several emitter calls, equivalent to `c(x); d(y)`

Non-blocking molecules

If `c` defined as above as `val c = m[Int]`, the emitter `c` is **non-blocking**. The emitter function call `c(123)` is non-blocking because it immediately returns a `Unit` value and does not wait for any reaction involving `c(123)` to actually start.

The non-blocking emitter `c` defined above will have type `M[Int]` and can be also created directly using the class constructor:

```
val c = new M[Int]("c")
```

Molecules carrying `Unit` values can be emitted using the syntax such as `a()` rather than `a(())`, which is also valid. The shorter syntax `a()` is provided as a purely syntactic convenience.

Blocking molecules

For a **blocking** molecule, the emitter call will block until a reaction can start that consumes that molecule.

A blocking emitter is defined like this,

```
val f = b[Int, String]
```

Now `f` is a blocking emitter that takes an `Int` value and returns a `String`.

Blocking emitters are values of type `B[T, R]`, which is a subtype of `T ⇒ R`. The emitter `f` could be equivalently defined by

```
val f = new B[Int, String]("f")
```

In this case, the user needs to provide the molecule name explicitly.

Once `f` is defined like this, an emitter call such as

```
val result = f(123)
```

will emit a molecule of sort `f` with value `123` into the soup.

After emitting `f(123)`, the process will become blocked until some reaction starts, consumes this molecule, and performs a **reply action**.

Since the type of `f` is `B[Int, String]`, the reply action must pass a `String` value to the reply function:

```
go { case c(x) + f(y, r) ⇒
  val replyValue = (x + y).toString
  r(replyValue)
}
```

The reply action consists of calling the **reply emitter** `r` with the reply value as its argument.

Only after the reply action, the process that emitted `f(123)` will become unblocked and the statement `val result = f(123)` will be completed. The variable `result` will become equal to the string value that was sent as the reply.

Remarks about the semantics of `Chymyst`

- Emitted molecules such as `c(123)` are *not* Scala values. Emitted molecules cannot be stored in a data structure or passed as arguments to functions. The programmer has no direct access to the molecules in the soup, apart from being able to emit them. But emitters *are* ordinary, locally defined Scala values and can be manipulated as any other Scala values. Emitters are functions whose `apply` method has the side effect of emitting a new copy of a molecule into the soup.
- Emitters are local values of type `B[T, R]` or `M[T]`. Both types extend the trait `MolEmitter` as well as the type `T ⇒ R`. Blocking molecule emitters are of type `B[T, R]`, non-blocking of type `M[T]`.
- Reactions are local values of type `Reaction`. Reactions are created using the function `go` with the syntax `go { case ... ⇒ ... }`.
- Only one `case` clause can be used in each reaction. It is an error to use several `case` clauses, or case clauses that do not match on input molecules, such as `go { case x ⇒ }`. The only exception is "static reactions" described later.
- Reaction sites are immutable values of type `ReactionSite`. These values are not visible to the user: they are created in a closed scope by the `site(...)` call. The `site(...)` call activates all the reactions at that reaction site.
- Molecule emitters are immutable after all reactions have been activated where these molecules are used as inputs.
- When emitted into the soup, molecules gather at their reaction sites. Reaction sites proceed by first deciding which input molecules can be consumed by some reactions; this decision involves the chemical sorts of the molecules as well as any pattern matching and guard conditions that depend on molecule values. When suitable input molecules are found and a reaction is chosen, the input molecules are atomically removed from the soup, and the reaction body is executed.
- The reaction body can emit one or more new molecules into the soup. The code can emit new molecules into the soup at any time and from any code, not only inside a reaction body.
- When enough input molecules are present at a reaction site so that several alternative reactions can start, is not possible to decide which reactions will proceed first, or which molecules will be consumed first. It is also not possible to know at what time reactions will start. Reactions and molecules do not have priorities and are not ordered in the soup. When determinism is required, it is the responsibility of the programmer to define the chemical laws such that the behavior of the program is deterministic. (This is always possible!)
- All reactions that share some *input* molecule must be defined within the same reaction site. Reactions that share no input molecules can (and should) be defined in separate reaction sites.

Chemical designations of molecules vs. molecule names vs. local variable names

Each molecule has a name — a string that is used in error messages and for debugging. The molecule's name must be specified when creating a new molecule emitter:

```
val counter = new M[Int]("counter")
```

Most often, we would like the molecule's name to be the same as the name of the local variable, such as `counter`, that holds the emitter. When using the `m` macro, specifying names in this way becomes automatic:

```
val counter = m[Int]
counter.name == "counter" // true
```

Each molecule also has a specific chemical designation, such as `sum`, `counter`, and so on. These chemical designations are not actually strings `"sum"` or `"counter"`. The names of the local variables are chosen purely for the programmer's convenience.

Rather, the chemical designations are the *object identities* of the molecule emitters. We could define an alias for a molecule emitter, for example like this:

```
val counter = m[Int]
val q = counter
```

This code will copy the molecule emitter `counter` into another local value `q`. However, this does not change the chemical designation of the molecule, because `q` will be a reference to the same JVM object as `counter`. The emitter `q` will emit the same molecules as `counter`; that is, molecules emitted with `q(...)` will react in the same way and in the same reactions as molecules emitted with `counter(...)`.

(This is similar to how chemical substances are named in ordinary language. For example, `NaCl`, "salt" and "sodium hydrochloride" are alias names for the same chemical substance.)

The chemical designation of the molecule specifies two aspects of the concurrent program:

- what other molecules (i.e. what other chemical designations) are required to start a reaction with this molecule, and at which reaction site;
- what computation will be performed when this molecule and all the other required input molecules are available.

When a new reaction site is defined, certain molecules become bound to that site. (These molecules are the inputs of reactions defined at the new site.) If a reaction site is defined within a local scope of a function, new molecule emitters and a new reaction site will be created *every time* the function is called. Since molecules internally hold a reference to their reaction site, each function call will create *chemically unique* new molecules, in the sense that these molecules will react only with other molecules defined in the same function call.

As an example, consider a function that defines a simple reaction like this:

```
def makeLabeledReaction() = {
  val begin = m[Unit]
  val label = m[String]

  site(
    go { case begin(_) + label(labelString) ⇒ println(labelString) }
  )

  (begin, label)
}

// Let's use this function now.

val (begin1, label1) = makeLabeledReaction() // first call
val (begin2, label2) = makeLabeledReaction() // second call
```

What is the difference between the new molecule emitters `begin1` and `begin2`? Both `begin1` and `begin2` have the name `"begin"`, and they are of the same type `M[Unit]`. However, they are *chemically* different: `begin1` will react only with `label1`, and `begin2` only with `label2`. The molecules `begin1` and `label1` are bound to the reaction site created by the first call to `makeLabeledReaction()`, and this reaction site is different from that created by the second call to `makeLabeledReaction()`.

For example, suppose we emit `label1("abc")` and `begin2()`. We have emitted a molecule named `"label"` and a molecule named `"begin"`. However, these molecules will not start any reactions, because they are bound to different reaction sites.

```
label1("abc") + begin2() // no reaction started!
```

After running this code, the molecule `label1("abc")` will be waiting at the first reaction site for its reaction partner, `begin1()`, while the molecule `begin2()` will be waiting at the second reaction site for its reaction partner, `label2(...)`. If we now emit, say, `label2("abc")`, the reaction with the molecule `begin2()` will start and print `"abc"`.

```
label1("abc") + begin2() // no reaction started!
label2("abc") // reaction with begin2() starts and prints "abc"
```

We see that `begin1` and `begin2` have different chemical designations because they enter different reactions. This is so despite the fact that both `begin1` and `begin2` are defined by the same code in the function `makeLabeledReaction()`. Since the code creates a new emitter value every time, the JVM object identities of `begin1` and `begin2` are different.

The chemical designations are independent of molecule names and of the variable names used in the code. For instance, we could (for whatever reason) create aliases for `label2` and `begin2` and write code like this:

```
val (begin1, label1) = makeLabeledReaction() // first call
val (begin2, label2) = makeLabeledReaction() // second call
val (x, y, p, q) = (begin1, label1, begin2, label2) // make aliases

y("abc") + p() // Same as label1("abc") + begin2() — no reaction started!
q("abc") // Same as label2("abc") — reaction starts and prints "abc"
```

In this example, the values `x` and `begin1` refer to the same molecule emitter, thus they have the same chemical designation. For this reason, the calls to `x()` and `begin1()` will emit copies of the same molecule. As we already discussed, the molecule emitted by `x()` will have a different chemical designation from that emitted by `begin2()`.

In practice, it is of course advisable to choose meaningful local variable names.

To summarize:

- each molecule has a chemical designation, a reaction site to which the molecule is bound, a name, and a molecule emitter (usually assigned to a local variable)
- the chemical reactions started by molecules depend only on their chemical designations, not on names
- the molecule's name is only used in debugging messages
- the names of local variables are only for the programmer's convenience
- reaction sites defined in a local scope are new and unique for each time a new local scope is created
- molecules defined in a new local scope will have a new, unique chemical designation and will be bound to the new unique reaction site

The type matrix of molecule emission

Let us consider what *could* theoretically happen when we call an emitter.

The emitter call can be either blocking or non-blocking, and it could return a value or return no value. Let us write down all possible combinations of emitter call types as a “type matrix”.

To use a concrete example, we assume that `c` is a non-blocking emitter of type `M[Int]` and `f` is a blocking emitter of type `B[Unit, Int]`.

	blocking emitter	non-blocking emitter
value is returned	`val x: Int = f()`	?
no value returned	?	`c(123)` // side effect

So far, we have seen that blocking emitters return a value, while non-blocking emitters don't. There are two more combinations that are not yet used:

- a blocking emitter that does not return a value
- a non-blocking emitter that returns a value

The `Chymyst` library implements both of these possibilities as special features:

- a blocking emitter can *time out* on its call and fail to return a value;
- some non-blocking emitters have a “volatile reader” (see below) that provides read-only access to the last known value of the molecule.

With these additional features, the type matrix of emission is complete:

	blocking emitter	non-blocking emitter
value is returned:	`val x: Int = f()`	`val x: Int = c.volatileValue`
no value returned:	(timeout was reached)	`c(123)` // side effect

We will now describe these features in more detail.

Timeouts for blocking emitters

By default, a blocking emitter will emit a new molecule and block until a reply action is performed for that molecule by some reaction. If no reaction can be started that consumes the blocking molecule, its emitter will block and wait indefinitely.

`Chymyst` allows us to limit the waiting time to a fixed timeout value. Timeouts are implemented by the method `timeout()()` on the blocking emitter:

```
val f = b[Unit, Int]
// create a reaction site involving `f` and other molecules:
site(...)

// call the emitter `f` with 200ms timeout:
val x: Option[Int] = f.timeout()(200 millis)
```

The first argument of the `timeout()()` method is the value carried by the emitted molecule. In this example, this value is empty since `f` has type `B[Unit, Int]`. The second argument of the `timeout()()` method is the duration of the delay.

The `timeout()()` method returns a value of type `Option[R]`, where `R` is the type of the blocking molecule's reply value. In the code above, if the emitter received a reply value `v` before the timeout expired then the value of `x` will become `Some(v)`.

If the emitter times out before a reply action is performed, the value of `x` will be `None` and the blocking molecule `f()` will be *removed* from the soup. If the timeout occurred because no reaction started with `f()`, which is the usual reason, the removal of `f()` makes sense because no further reactions should try to consume `f()` and reply.

A less frequent situation is when a reaction already started, consuming `f()` and is about to reply to `f()`, but the waiting process just happened to time out at that very moment. In that case, sending a reply to `f()` will have no effect.

Is the timeout feature required? The timeout functionality can be simulated, in principle, using the “First Reply” construction. However, this construction is cumbersome and will sometimes leave a thread blocked forever, which is undesirable from the implementation point of view. For this reason, `Chymyst` implements the timeout functionality as a special primitive feature available for blocking molecules.

In some cases, the reaction that sends a reply to a blocking molecule needs to know whether the waiting process has already timed out. For this purpose, `Chymyst` defines the reply emitters as functions `R ⇒ Boolean`. The return value is `true` if the waiting process has received the reply, and `false` otherwise.

Static molecules

It is often useful to ensure that exactly one copy of a certain molecule is initially present in the soup, and that no further copies can be emitted unless one is first consumed. Such molecules are called **static**. A static molecule `s` must have reactions only of the form `s + a + b + ... → s + c + d + ...`, — that is, reactions that consume a single copy of `s` and then also emit a single copy of `s`.

An example of a static molecule is the "asynchronous counter" molecule `c()` with reactions that we have seen before:

```
go { case c(x) + d(_) ⇒ c(x - 1) }
go { case c(x) + i(_) ⇒ c(x + 1) }
go { case c(x) + f(_, r) ⇒ c(x) + r(x) }
```

These reactions treat `c()` as a static molecule because they first consume and then emit a single copy of `c()`.

Static molecules are a frequently used pattern in chemical machine programming. `Chymyst` provides special features for static molecules:

- Only non-blocking molecules can be declared as static.
- Static molecules are emitted directly from a reaction site definition, by special static reactions that run only once. In this way, static molecules are guaranteed to be emitted once and only once, before any other reactions are run and before any molecules are emitted to that reaction site.
- Static molecules have "volatile readers".

In order to declare a molecule as static, the users of `Chymyst` must write a reaction that has no input molecules but emits some output molecules. Such reactions are automatically recognized by `Chymyst` as **static reactions**:

```
site (
// This static reaction declares a, c, and q to be static molecules and emits them.
    go { case _ ⇒ a(1) + c(123) + q() },
// Now define some more reactions that consume a, c, and q.
    go { case a(x) + ... ⇒ ???; a(y) } // etc.
)
```

The reaction `go { case _ ⇒ a(1) + c(123) + q() }` emits three output molecules `a()`, `c()`, and `q()` but has a wildcard instead of input molecules. `Chymyst` detects this and marks the reaction as static. The output molecules are also declared as static.

A reaction site can have several static reactions. Each non-blocking output molecule of each static reaction is automatically declared a **static molecule**.

The reaction sites will run their static reactions only once, at the time of the `site(...)` call itself, and on the same thread that calls `site(...)`. At that time, the reaction site still has no molecules present. Static molecules will be the first ones emitted into that reaction site.

Constraints on using static molecules

Declaring a molecule as static can be a useful tool for avoiding errors in chemical machine programs because the usage of static molecules is tightly constrained:

- A static molecule can be emitted only by a reaction that consumes it, or by the static reaction that defines it.
- A reaction may not consume more than one copy of a static molecule.
- A reaction that consumes a static molecule must also have some non-static input molecules.
- It is an error if a reaction consumes a static molecule but does not emit it back into the soup, or emits it more than once.
- It is also an error if a reaction emits a static molecule it did not consume, or if any other code emits additional copies of the static molecule at any time.
- A reaction may not emit static molecules from within a loop or within function calls.

These restrictions are intended to maintain the semantics of static molecules. Application code that violates these restrictions will cause an "early" run-time error - that is, an exception thrown by the `site()` call before any reactions can run at that reaction site.

Volatile readers for static molecules

When a static molecule exists only as a single copy, its value works effectively as a mutable cell: the value can be modified by reactions, but the cell cannot be destroyed. So it appears useful to have a read-only access to the value in the cell.

This is implemented as a **volatile reader** — a function of type `⇒ T` that fetches the value carried by that static molecule when it was most recently emitted. Here is how volatile readers are used:

```
val c = m[Int]
site(
  go { case c(x) + incr(_) ⇒ c(x + 1) },
  go { case _ ⇒ c(0) } // emit `c(0)` and declare it as static
)

val readC: Int = c.volatileValue // initially returns 0
```

The volatile reader is thread-safe (can be used from any reaction without blocking any threads) because it provides a read-only access to the value carried by the molecule.

The value of a static molecule, viewed as a mutable cell, can be modified only by a reaction that consumes the molecule and then emits it back with a different value. If the volatile reader is called while that reaction is being run, the reader will return the previous known

value of the static molecule, which is probably going to become obsolete very shortly. We call the volatile reader “volatile” for this reason: it returns a value that could change at any time.

The functionality of a volatile reader is similar to an additional reaction with a blocking molecule `f` that reads the current value carried by `c`:

```
go { case c(x) + f(_, reply) ⇒ c(x) + reply(x) }
```

Calling `f()` returns the current value carried by `c()`, just like the volatile reader does. However, the call `f()` may block for an unknown time if `c()` has been consumed by a long-running reaction, or if the reaction site happens to be busy with other computations. Even if `c()` is immediately available in the soup, running a new reaction requires an extra scheduling operation. Volatile readers provide fast read-only access to values carried by static molecules.

This feature is restricted to static molecules because it appears to be useless if we read the last emitted value of a molecule that has many different copies emitted in the soup.

Exercise: concurrent divide-and-conquer with cancellation

Implement a (blocking) function that finds the smallest element in an integer array, using a concurrent divide-and-conquer method. If the smallest element turns out to be negative, the computation should be aborted as early as possible, and `0` should be returned as the final result.

Use a static molecule with a volatile reader to signal that the computation needs to be aborted early.

Constraints of the Chemical Machine (and how to overcome them)

While designing the chemical laws for an application, we need to keep in mind that the chemical paradigm limits what we can do. As we will come to realize, these constraints are for our own good!

Absence of molecules is undetectable

We cannot detect the *absence* of a given non-blocking molecule, say `a(1)`, in the soup. This seems to be a genuine limitation of the chemical machine paradigm.

It seems that this limitation cannot be lifted by any clever combinations of blocking and non-blocking molecules; perhaps this can be even proved formally, but I haven't tried learning the formal tools for that. I just tried to implement this but could not find the right combination of reactions.

How could we possibly detect the absence of a given molecule, say `a(x)` with any value `x`? We can emit a blocking molecule that reacts with `a(x)`. If `a(x)` is absent, the emission will block. So the absence of `a(x)` in the soup can be translated into blocking of a function call. However, no programming language is able to detect whether a function call has been blocked, because the function call is by definition a blocking call! All we can do is to detect whether the function call has returned within a given time, but here we would like to return instantly with the information that `a` is present or absent.

Suppose we define a reaction that consumes the molecule `a()`. Even if we establish that this reaction did not start within a certain time period, we cannot conclude that `a()` is absent in the soup at that time! It could happen that `a()` was present but got involved in some other reactions and was consumed by them, or that `a()` was present but the computer's CPU was simply so busy that our reaction could not yet start and is still waiting in a queue.

Another feature would be to introduce “inhibiting” conditions on reactions: a certain reaction can start when molecules `a` and `b` are present but no molecule `c` is present. However, it is not clear that this extension of the chemical paradigm would be useful. The reactions with “inhibiting” conditions will be unreliable because they will sometimes run and sometimes not run, depending on exactly when some molecules are emitted. Since the programmer cannot control the duration of time taken by reactions, it seems that “inhibiting” conditions simply lead to a kind of indeterminism that the programmer cannot control at all.

Since we can expect molecules to be emitted at random and unpredictable times by concurrently running processes, it is always possible that a certain molecule is, at a given time, not present in the soup but is about to be emitted by some reaction because the reaction task is already waiting in the scheduler's queue. If we added a feature to the chemical machine that explicitly detects the absence of a molecule, we would merely make the program execution unreliable while giving ourselves an illusion of control.

With its present design, the chemical machine forces the programmer to design the chemical laws in such a way that the result of the execution of the program is the same even if random delays were inserted at any point when a molecule is emitted or a reaction is started.

No remote pooling of molecules

Chemical soups running as different processes (either on the same computer or on different computers) are completely separate and cannot be pooled.

It could be useful to be able to connect many chemical machines together, running perhaps on different computers, and to pool their individual soups into one large “common soup”. Our program should then be able to emit lots of molecules into the common pool and thus organize a massively parallel, distributed computation, without worrying about which CPU computes what reaction.

Some implementations of the chemical machine, notably JoCaml, provide a facility for sending molecules from one chemical soup to another. However, in order to organize a distributed computation, we would need to split the tasks explicitly between the participating soups. The organization and supervision of distributed computations, the maintenance of connections between machines, the handling of disconnections — all this remains the responsibility of the programmer and is not handled automatically by JoCaml.

In principle, a sufficiently sophisticated runtime engine could organize a distributed computation completely transparently to the programmer. It remains to be seen whether it is feasible and/or useful to implement such a runtime engine.

Chemistry is immutable

Reactions and reaction sites are immutable. It is impossible to add more reactions at run time to an existing reaction site. This limitation is enforced in `Chymyst` by making reaction sites immutable and invisible to the user.

After a reaction site declares a certain molecule as an input molecule for some reactions, it is impossible to add further reactions consuming that molecule. It is also impossible to remove reactions from reaction sites, or to disable reactions. The only way to stop certain reactions from running is to refrain from emitting some input molecules required by these reactions.

However, it is possible to declare a reaction site with reactions computed *at run time*. Since reactions are local values (as are molecule emitters), users can first create any number of reactions and store these reactions in an array, before declaring a reaction site with these reactions. Once all desired reactions have been assembled, users can declare a reaction site that includes all the reactions from the array.

As an (artificial) example, consider the following pattern of reactions:

```
val finished = m[Unit]
val a = m[Int]
val b = m[Int]
val c = m[Int]
val d = m[Int]

site(
  go { case a(x) ⇒ b(x + 1) },
  go { case b(x) ⇒ c(x + 1) },
  go { case c(x) ⇒ d(x + 1) },
  go { case d(x) ⇒ if (x > 100) finished() else a(x + 1) }
)

a(10)
```

When this is run, the reactions will cycle through the four molecules `a`, `b`, `c`, `d` while incrementing the value each time, until the value 100 or higher is reached by the molecule `d`.

Now, suppose we need to write a reaction site where we have `n` molecules and `n` reactions instead of just four, where `n` is a run-time parameter. Since molecule emitters and reactions are local values, we can simply create them and store in a data structure:

```
val finished = m[Unit]
val n = 100 // `n` is computed at run time

// sequence of molecule emitters:
val emitters = (0 until n).map( i ⇒ new M[Int](s"a_$i"))
// this is equivalent to declaring:
// val emitters = Seq(
//    new M[Int]("a_0"),
//    new M[Int]("a_1"),
//    new M[Int]("a_2"),
//    ...
// )

// array of reactions:
val reactions = (0 until n).map{ i ⇒
  // create the i-th reaction with index
  val iNext = if (i == n - 1) 0 else i + 1
  val a = emitters(i) // We must define molecule emitters `a`
  val aNext = emitters(iNext) // and `aNext` as explicit local values,
  go { case a(x) ⇒ // because `case emitters(i)(x)` won't compile.
    if (i == n - 1 && x > 100) finished() else aNext(x + 1)
  }
}

// write the reaction site
site(reactions:_*)

// emit the first molecule
emitters(0)(10)
```

Pipelined molecules

The chemical machine paradigm does not enforce any particular order on reactions or molecules. However, it is often necessary in practical applications to ensure that e.g. requests are processed in the order they are received. If each request is represented by a newly emitted molecule, we will be required to ensure that molecules start their reactions in the order of emission.

In principle, this can be implemented by, say, attaching an extra timestamp to each molecule and enforcing the order of molecule consumption via guard conditions on all relevant reactions. But this method of implementation is inefficient because the reaction scheduler will have to enumerate all the present molecules, looking for a molecule with the right timestamp. The programmer will be also burdened with extra bookkeeping, and the code will become less declarative.

For this reason, `Chymyst` has special support for **pipelined molecules** - that is, molecules that are always consumed in the exact order they were emitted.

For each pipelined molecule, `Chymyst` allocates an (ordered) queue that stores molecule instances in the order they were emitted. Whenever a reaction consumes a pipelined molecule, it is always the head of the queue that is consumed.

Pipelined molecules are detected automatically by `Chymyst` according to the conditions we will describe now. First we need to consider whether the chemical machine semantics is compatible with pipelined molecules at all.

It turns out that, in a given chemical program, some molecules may be made pipelined without adverse effects, while others cannot be made pipelined. The following example illustrates why this is so. In this example, several reactions consume the same molecule but impose different guard conditions on the molecule value:

```
site(
  go { case c(0) + a(y) ⇒ ... },
  go { case c(y) if y > 0 ⇒ a(y) }
)
```

Suppose the molecule `c()` is implemented via an ordered queue, and we emit `c(0)`, `c(1)`, `c(2)` in this order. The presence of `c(0)` at the head of the queue means that a reaction consuming `c(0)` must run first.

Thus, the presence of `c(0)` at the head of the queue will prevent any other copy of `c()` from being consumed, until a molecule `a()` becomes available so that `c(0)` may be consumed and removed from the head of the queue. But `a()` is emitted only by the second reaction, which will never run since `c(1)` is not at the head of the queue.

So, implementing `c()` as a pipelined molecule will create a deadlock in this program. Programs like this one will work correctly only if reactions are allowed to consume the `c()` molecules out of order.

However, the molecule `a()` can be made pipelined, since there are no conditions on its value, and the available instances of `a()` can be consumed in any order without creating deadlocks.

Many chemical programs can be implemented with pipelined molecules. As another example, consider the "counter",

```
site(
  go { case c(x) + incr(_) ⇒ c(x + 1) },
  go { case c(x) + decr(_) if x > 0 ⇒ c(x - 1) }
)
```

In this program, the molecules `incr()` and `decr()` can be made pipelined without any adverse effects.

So we find that, in a given chemical program, some molecules can be made pipelined while others cannot. `Chymyst` performs static (early run-time) analysis of the user's code and automatically assigns ordered queues to all molecules that could admit pipelined semantics without creating problems.

For a molecule `c()` to be pipelined, the reactions consuming `c()` must be such that the chemical machine only needs to inspect a single copy of the molecule `c()` in order to be able to select correctly the next reaction to run.

One use case for pipelined molecules is implementing a chain of asynchronous processing stages. Suppose that we need to process initial data by using transformations `x ⇒ f(x)`, then `x ⇒ g(x)`, etc., where each stage takes a long time. We would like to run all these stages concurrently. At the same time, we would like to retain the order in which the data goes through the processing stages.

The following program implements an example of this kind of computation. Each processing stage maintains local state (denoted by `s`), which is updated to a new value `newS` while the result is computed and passed on to the next stage.

```
val input1 = m[Int]
val stage1 = m[Int]
val input2 = m[Int]
val stage2 = m[Int]
val input3 = m[Int]
val stage3 = m[Int]
val output = m[Int]

site(
  go { case input1(x) + step1(s) ⇒
    val newS = p(x, s)
    val result = f(x, s)
    stage1(newS) + input2(result)
  },
  go { case input2(x) + step2(s) ⇒
    val newS = q(x, s)
    val result = g(x, s)
    stage2(newS) + input3(result)
  },
  go { case input3(x) + step3(s) ⇒
    val newS = r(x, s)
    val result = h(x, s)
    stage3(newS) + output(result)
  }
)

stage1(0) + stage2(0) + stage3(0)

(1 to 10000).foreach(i ⇒ input1(i))
```

In this program, all three kinds of "input" molecules are pipelined, which means that they are consumed in the same order as they are emitted. This will create a linearly ordered data stream even if we emit a large number of `input1()` molecules at once.

It is important to note that having each of the "input" pipelined will not be sufficient for maintaining the linear order of the data stream.

If we initially emit several copies of `stage1()`, several instances of the first stage could run in parallel. Typically, the parallel-running instances will not finish in the same order they started, so the `input2()` molecules will not be emitted in the same order as `input1()`. The only way to prevent this from happening is to assign a custom single-thread pool to each of the stage reactions.

Therefore, we have two ways of building a linearly ordered data stream:

- use static molecules or other single-copy molecules, together with pipelined molecules
- use single-thread pools, together with pipelined molecules

Detecting pipelined molecules

In the examples we have seen, there were no special annotations or syntax for pipelined molecules. This is so because `Chymyst` determines automatically which molecules should be made pipelined, in a given chemical program. Here is how this is decided.

Let us assume that a molecule `c()` has value type `T`.

There may exist values `x: T` such that no reactions will *ever* consume the instance `c(x)` with that `x` -- neither now, nor in the future. (This happens, for example, if `x` is such that no pattern or guard condition is ever satisfied in any of the available reactions. Molecule instances `c(y)` with some `y != x` might still be potentially consumed now or later.) Let us call these values of `x` **ignorable** values for the molecule `c()`. Molecule instances `c(x)` that carry an ignorable value `x` will never be consumed by any reactions and can be deleted from the soup immediately.

Let us now examine the main assumption about pipelined molecules: For a molecule `c()` to be pipelined, the chemical program must be such that the next reaction consuming `c()` can be always found (with no deadlocks) by examining *only one* instance `c(x)` of the molecule.

Two properties of the chemical program immediately follow from this requirement:

- (Property 1.) Every reaction may consume at most one instance of `c()`. (No reaction should consume multiple input `c()` molecules, e.g. `go { c(x) + c(y) ⇒ ... }`.)
- (Property 2.) If, for some non-ignorable value `x`, the molecule instance `c(x)` cannot be immediately consumed by any reaction at this time, no other molecule instance `c(y)` with *any other* non-ignorable `y != x` can be immediately consumed by any reaction either. (If some reaction could consume `c(y)`, the program would become deadlocked by the presence of `c(x)` at the top of the queue.)

Property 2 is somewhat difficult to reason about, when formulated in this way. By using Boolean logic, this property can be simplified and formulated as an easier property to be checked for all reactions consuming `c()`. A mathematical derivation of the simplified property is in the next subsection.

To formulate the simplified property, let us assume that `c(x)` is at the head of the queue, and let us ask whether a molecule instance `c(x)` can be consumed by a certain reaction, say `go { c(x) + d(y) + e(z) if g(x, y, z) ⇒ ... }`. This reaction can start if

- a molecule instance `d(y)` is present with some `y`,
- a molecule instance `e(z)` is present with some `z`,
- the values `y` and `z` satisfy the guard condition `g(x, y, z)`.

We can write these conditions symbolically as a single Boolean formula

`HAVE(d(y)) && HAVE(e(z)) && g(x, y, z)`.

Repeating the same consideration for each reaction consuming `c()`, we obtain a certain set of Boolean formulas of this kind, with various guard conditions and other molecules.

Now, the conditions for `c()` to be pipelined are that

- each reaction's Boolean formula can be identically rewritten as a simple Boolean *conjunction* of the form `p(x) && q(y, z, ...)`, where
- the Boolean function `p(x)` involves only the value `x` and must be the same for all reactions, while `q(...)` could be different for each reaction, involving the presence of any other molecules and their specific values, but independent of `x`.

In other words, the condition for `c(x)` to be consumed by any reaction must depend on the value `x` in a way that does not involve other molecules.

To check whether any molecule `c()` satisfies this condition within a given chemical program, `Chymyst` goes through all reactions that consume `c(x)`, determines the Boolean formula for that reaction, and tries to simplify that formula into a conjunction of the form `p(x) && q(y, z, ...)`. If this succeeds with `p(x)` being *the same for all reactions*, `Chymyst` determines that the molecule `c()` can be pipelined.

`Chymyst` will then assign the `.isPipelined` property of the molecule emitter to `true` for that molecule. In debug output, the names of pipelined molecules will be suffixed with `/P`.

Here is an example of a reaction site where some molecules are pipelined but others are not:

```
site(
  go { case a(x) if x > 0 + e(_) ⇒ ... },
  go { case a(x) + c(y) if x > 0 ⇒ ... },
  go { case d(z) + c(y) if y > 0 ⇒ ...}
)
```

In this program, `a(x)` can be consumed if and only if `x > 0`. This condition is the same for the two reactions that consume `a()`. Therefore, the condition for consuming `a(x)` has the form `x > 0 && q(...)` for some Boolean function `q()` that does not depend on `x`. `Chymyst` will determine that `a()` can be pipelined.

The situation with the molecule `c()` is different: The second reaction accepts `c(y)` with any `y` but the third reaction requires `y > 0`. Therefore the Boolean condition for consuming `c(y)` has the form

`q1(HAVE(a(x)), x) || y > 0 && Q2(HAVE(d(z)))`

This formula cannot be decomposed into a conjunction of the form `p(y) && q(x, z, ...)` So the molecule `c()` cannot be pipelined in this reaction site.

Finally, the molecule `d()` is pipelined because it is consumed without imposing any conditions on its value. Such molecules can be always pipelined.

Derivation of the simplified condition

This subsection shows a formal derivation of the pipelined condition using Boolean logic. Readers not interested in this theory may skip to next section.

Consider a set of reactions, each consuming a single instance of `c(x)`. For each reaction `r`, we have a Boolean function of the form `f_r(x, y, z, HAVE(d(y)), HAVE(e(z)), ...)` that depends on a number of other variables.

Here, the symbolic expressions such as `HAVE(d(y))` are understood as simple Boolean variables. Values of these variables, as well as values of `y`, `z`, etc., describe the current population of molecules in the soup, excluding the molecule `c(x)`. For brevity, we denote all these "external" variables by `E` (these variables do not include `x`).

The condition for a molecule `c(x)` to be consumed by any reaction is the Boolean disjunction of all the per-reaction functions `f_r(x, E)`. Let us denote this disjunction by `F(x, E)`:

`F(x, E) =` f _{r_1} (x, E) || f _{r_2} (x, E) || ... || f _{r_n} (x, E)`.

We will now show that the complicated requirements of Property 2 from the previous section are equivalent to the single requirement that

`F(x, E) = p(x) && q(E)`,

for some Boolean functions `p` and `q`.

Property 2 states that, for any `x` and at any time (i.e. for any `E`), one of the three conditions must hold:

1. The value `x` is ignorable.
2. The molecule `c(x)` can be consumed immediately by some reaction.
3. The molecule `c(x)` cannot be consumed immediately by any reaction, but also no other molecule instance `c(y)` with any `y != x` could be consumed immediately by any reaction, if that `c(y)` were present in the soup.

Let us formulate these conditions in terms of the function `F(x, E)`, where `x` and `E` describe the soup at the present time:

1. `∀ E1 : !F(x, E1)`
2. `F(x, E)`
3. `∀ x1 : !F(x1, E)`

The disjunction of these three Boolean formulas must be `true` for all `x` and `E`:

`∀ x : ∀ E : (∀ E1 : !F(x, E1)) || F(x, E) || (∀ x1 : !F(x1, E))`.

We will use some tricks of Boolean algebra to prove that this formula is identically equivalent to

`∀ x : ∀ E : F(x, E) == p(x) && q(E)`

where `p` and `q` are suitably defined Boolean functions.

To prove this, we will identically transform the expression under the quantifiers `∀ x : ∀ E :` into the form `F(x, E) == p(x) && q(E)`.

Note that, in Boolean logic, `a == b` is the same as `a && b || (!a && !b)`. Thus our goal is to derive

(**) `∀ x : ∀ E : F(x, E) && p(x) && q(E) || (!F(x, E) && !(p(x) && q(E)))`.

The derivation proceeds in three steps.

(1) We have the identity `!(∀ x : !f(x)) = ∃ x : f(x)` for any Boolean function `f`.

Therefore, we can rewrite

- `(∀ E1 : !F(x, E1)) = !(∃ E1 : F(x, E1))`
- `(∀ x1 : !F(x1, E)) = !(∃ x1 : F(x1, E))`

We also note that these two expressions are functions of `x` and of `E` respectively.

Let us introduce names for these functions:

- Define `p(x) = ∃ E1 : F(x, E1)`.
- Define `q(E) = ∃ x1 : F(x1, E)`.

Then we can rewrite the original expression under the quantifiers as

`(∀ E1 : !F(x, E1)) || F(x, E) || (∀ x1 : !F(x1, E))`

= `F(x, E) || !p(x) || !q(E) = F(x, E) || !(p(x) && q(E))`.

(2) Another Boolean identity that holds for any Boolean functions `A` and `B` is

`A || !B = (A && B) || !B`.

We use this identity to rewrite the result of step (1) as

`F(x, E) || !(p(x) && q(E)) = F(x, E) && p(x) && q(x) || !(p(x) && q(E))` .

(3) We observe that, for any Boolean function `h(x)` and for any `a` ,

`(∀ x : h(x)) = h(a) && (∀ x : h(x))` .

Adding an extra conjuction with `h(a)` does not change the value of `(∀ x : h(x))` : If `(∀ x : h(x))` is true, it means that `h(x)` holds for all possible values of `x` , including `x = a` .

Using this transformation, we find that

- `!p(x) = (∀ E1 : !F(x, E1)) = !F(x, E) && (∀ E1 : !F(x, E1)) = !F(x, E) && !p(x)`
- `!q(E) = (∀ x1 : !F(x1, E)) = !F(x, E) && (∀ x1 : !F(x1, E)) = !F(x, E) && !q(E)`

It follows that

`!(p(x) && q(E)) = !p(x) || !q(E) = !F(x, E) && !p(x) || !F(x, E) && !q(E)`

= `!F(x, E) && (!p(x) || !q(E)) = !F(x, E) && !(p(x) && q(E))` .

We use this identity to rewrite the result of step (2) as

`F(x, E) && p(x) && q(x) || !(p(x) && q(E))`

= `(F(x, E) && p(x) && q(E)) || (!F(x, E) && !(p(x) && q(E)))`

= `(F(x, E) == p(x) && q(E))` .

This is precisely the expression (**). Therefore, we find that for all `x` and `E` the Boolean function `F(x, E)` is equal to the conjunction `p(x) && q(E)` .

Note that we have proved the decomposition of `F(x, E)` rather than each of the per-reaction functions `f r (x, E)` . If each of the per-reaction functions is decomposable with the same `p(x)` , their disjunction `F(x, E)` is also decomposable; but the converse does not hold.

`Chymyst` checks that each per-reaction Boolean function is decomposed as a conjunction `p(x) && ...` with the same `p(x)` . Therefore, the condition actually checked by `Chymyst` is a sufficient (but not a necessary) condition for the molecule to be pipelined.

The result is that `Chymyst` might, in rare cases, fail to determine that some molecules could be pipelined. The lack of pipelining is safe and will, at worst, lead to some degradation of performance.

It would have been nearly impossible to obtain `F(x, E)` symbolically and to implement the strict necessary condition, because `x` and `E` are variables with values of arbitrary types (e.g. `Int` , `Double` , `String` and so on), which are not easily described through Boolean algebra.

As an (admittedly artificial) example, consider the reaction site

```
site(
  go { case c(x) + d(y) if x >= 0 && y > 0 ⇒ ... },
  go { case c(x) + d(y) if x >= 0 && y < 1 ⇒ ... },
  go { case c(x) + d(y) if x < 0 ⇒ ... }
)
```

The Boolean condition for consuming `c(x)` has the form

```
HAVE(d(y)) && (x >= 0 && y > 0 || x >= 0 && y < 1 || x < 0)
```

This condition is equivalent to `HAVE(d(y))` because `y > 0 || y < 1` is equivalent to `true`. Thus, the molecule `c()` can be pipelined. However, `Chymyst` cannot deduce that `y > 0 || y < 1` is equivalent to `true` because simplifying numerical inequalities goes far beyond Boolean logic and may even be undecidable in some cases.

Reaction constructors

Chemical reactions are static: they are specified at compile time and cannot be modified at run time. `Chymyst` compiles with this limitation even though reactions in `Chymyst` are values created at run time. For instance, we can create an array of molecules and reactions, where the size of the array is determined at run time. We can easily define reactions for "dining philosophers" even if the number of philosophers is given at run time.

Nevertheless, reactions will not be activated until a reaction site is created as a result of calling `site()`, which can be done only once. After calling `site()`, we cannot add or remove reactions. We also cannot write a second reaction site using an input molecule that is already bound to a previous reaction site. So, we cannot modify the list of molecules bound to a reaction site, and we cannot modify the reactions that may start there. In other words, chemistry at a reaction site is immutable.

For this reason, reaction sites are static in an important sense that guarantees that chemical laws continue to work as designed, regardless of what the application code does at a later time. This feature allows us to design chemistry in a modular fashion. Each reaction site encapsulates some chemistry and monitors certain molecule emitters. When user code emits a molecule, say `c()`, the corresponding reaction site has already fixed all the reactions that could possibly start due to the presence of `c()`. Users can neither disable these reactions nor add another reaction that will also consume `c()`. In this way, users are guaranteed that the encapsulated chemistry will continue to work correctly.

Nevertheless, reactions can be defined at run time. There are several techniques we can use:

1. Define molecules whose values contain other molecule emitters, which are then used in reactions.
2. Incrementally define new molecules and new reactions, store them in data structures, and assemble a reaction site later.
3. Define a new reaction site in a function that takes arguments and returns new molecule emitters.
4. Define molecules whose values are functions that represent reaction bodies.

We already saw examples of using the first two techniques. Let us now talk about the last two in some more detail.

Reaction constructor as a function

Since molecule emitters are local values that close over their reaction sites, we can easily define a general “1-molecule reaction constructor” that creates an arbitrary reaction with a single input molecule.

```
def makeReaction[T](reaction: (M[T],T) ⇒ Unit): M[T] = {
  val a = new M[T]("auto molecule 1") // the name is just for debugging
  site( go { case a(x) ⇒ reaction(a, x) } )
  a
}
```

Since `reaction` is an arbitrary function, it can emit further molecules if needed. In this way, we implemented a “reaction constructor” that can create an arbitrary reaction involving one input molecule.

Similarly, we could create reaction constructors for more input molecules:

```
def makeReaction2[T1, T2](reaction: (M[T1], T1, M[T2], T2) ⇒ Unit): (M[T1], M[T2]) = {
  val a1 = new M[T1]("auto molecule 1")
  val a2 = new M[T1]("auto molecule 2")
  site( go { case a1(x1) + a2(x2) ⇒ reaction(a1, x1, a2, x2) } )
  (a1, a2)
}
```

Reaction constructor as a molecule

In the previous example, we have encapsulated the information about a reaction into a closure. Since molecules can carry values of arbitrary types, we could put that closure onto a molecule. In effect, this will yield a “universal molecule” that can define its own reaction.

(However, the reaction can have only one molecule as input.)

```
val u = new M[Unit ⇒ Unit]("universal molecule")
site( go { case u(reaction) ⇒ reaction() } )
```

To use this “universal molecule”, we need to supply a reaction body and put it onto the molecule while emitting. In this way, we can emit the molecule with different reactions.

```
val p = m[Int]
val q = m[Int]
// emit u(...) to make the reaction u(x) ⇒ p(123)+q(234)
u({ _ ⇒ p(123) + q(234) })
// emit u(...) to make the reaction u(x) ⇒ p(0)
u({ _ ⇒ p(0) })
```

This example is artificial and perhaps not very useful; it just illustrates some of the capabilities of the chemical machine.

It is interesting to note that the techniques we just described are not special features of the chemical machine. Rather, they follow naturally from embedding the chemical machine within a functional language such as Scala, which has local scopes and can treat functions as values. The same techniques will work equally well if the chemical machine were embedded in any other functional language.

Working with an external asynchronous APIs

We now consider the task of interfacing with an external library that does not use the chemical machine paradigm.

Suppose we are working with an external library, such as an HTTP or database client, that has an asynchronous API via Scala's `Future`s. In order to use such libraries together with `Chymyst`, we need to be able to pass freely between `Future`s and molecules. The `Chymyst` standard library provides a basic implementation of this functionality.

To use the `Chymyst` standard library, add `"io.chymyst" %% "chymyst-lab" % "latest.integration"` to your project dependencies and import `io.chymyst.lab._`

Attaching molecules to futures

The first situation is when the external library produces a future value `fut : Future[T]`, and we would like to automatically emit a certain molecule `f` when this `Future` is successfully resolved. This is as easy as doing a `fut.map{x ⇒ f(123)}` on the future. The library has helper functions that add syntactic sugar to `Future` in order to reduce boilerplate in the two typical cases:

- the molecule needs to carry the same value as the result value of the future: `fut & f`
- the molecule needs to carry a different but fixed value: `fut + f(123)`

Attaching futures to molecules

The second situation is when an external library requires us to pass a future value that we produce. Suppose we have a reaction that will eventually emit a molecule with a result value. We now need to convert the emission event into a `Future` value, resolving to that result value when the molecule is emitted.

This is implemented by the `moleculeFuture` method. This method will create at once a new molecule emitter and a new `Future` value. The new molecule will be already bound to a reaction site. We can then use the new molecule as output in our reactions.

```
import io.chymyst.lab._

val a = m[Int]

// emitting the molecule result(...) will resolve "fut"
val (result: M[String], fut: Future[String]) = moleculeFuture[String]

// define a reaction that will eventually emit "result(...)"
site( go { case a(x) ⇒ result(s"finished: $x") } )

// the external library takes our value "fut" and does something with it
ExternalLibrary.consumeUserFuture(fut)

// Now write some code that eventually emits `a` to start the reaction above.
```

The four levels of concurrency

Written by Sergei Winitzki

The words "concurrent programming", "asynchronous programming", and "parallel programming" are used in many ways and sometimes interchangeably. However, there do in fact exist different, inequivalent levels of complexity that we encounter when programming applications that run multiple tasks simultaneously. I will call these levels "parallel data", "acyclic dataflow", "cyclic dataflow", and "general concurrency".

As we will see, each level is a strict subset of the next.

Level 1: Parallel data

The main task here is to process a large volume of data more quickly, by splitting the data in smaller subsets and processing all subsets in parallel (on different CPU cores and/or on different machines).

The computation is sequential, and we could have computed the same results on a single processing thread, without any splitting. Typically, the fully sequential, single-thread computation would be too slow, and thus we are trying to speed it up by using multiple threads.

A typical data-parallel task is to produce a table of word counts in 10,000 different text files. This class of problems is solved by such frameworks as Scala's parallel collections, Hadoop, and Spark. The main difficulty for the implementers of these frameworks is to "parallelize" the task, that is, to split the task correctly into subtasks that are as independent as possible, and to run these subtasks in parallel as efficiently as possible.

Parallel data = applicative stream

From the type-theoretic point of view, the splitting of the large data set into small chunks is equivalent to replacing the whole data set by a parameterized container type `Stream[T]`, where `T` is the type of one chunk of data. For example, in Spark this parameterized container type is the `RDD[T]` type.

A parallel data framework provides a number of operations on the `Stream[T]` type, such as `map` or `filter`. These operations make this type into a functor. Usually, there is also an operation such as `map2` that applies a function of several arguments to several containers at once. This operation makes `Stream[T]` into an applicative functor.

However, importantly, there is no `flatMap` operation that would make `Stream[T]` into a monad. A data-parallel framework limits its expressive power to that of an applicative functor.

It is this limitation that makes data-parallel frameworks so effective in their domain of applicability. As is well known in the functional programming folklore, applicative functor operations are easily and efficiently parallelized, but the monadic `flatMap` operation is not easily parallelized.

Level 2: Acyclic dataflow

The main task here is to process a large volume of data that is organized as one or more data streams. Each element of a stream is a chunk of data that can be processed independently from other chunks. The streams form an acyclic graph that starts "upstream" at "source" vertices and flows "downstream", finally ending at "sink" vertices. Other vertices of the graph are processing steps that transform the chunks of data in some way. Usually, programmers want to achieve the maximum throughput (number of chunks consumed at "sources" and delivered to "sinks" per unit time).

The dataflow is *asynchronous*: each vertex of the graph can perform its computation and deliver a result to a next vertex, even though that vertex might be still busy with its own processing. Since the computations could take different amounts of time at different vertices, certain processing steps will have lower throughput and incur longer wait times for subsequent steps. To compensate for this, we could run the slower processing steps in parallel on several data chunks at once, while other steps may run sequentially on each chunk. Therefore, acyclic dataflow systems can be also called "asynchronous parallel streaming systems".

A typical use case of acyclic dataflow is to implement a high-throughput asynchronous Web server that can start responding to the next request long before the previous request is answered. This class of problems can be solved by using `Future[T]`, by asynchronous streaming frameworks such as Akka Streaming, scala/async, or FS2, and by various functional reactive programming (FRP) frameworks. The main difficulty for the implementers of these frameworks is to interleave the wait times on each thread as much as possible and to avoid wasting CPU cycles when some threads are blocked.

As in the case of data-parallel computations, the acyclic dataflow computation is still equivalent a sequential computation, in the following sense: We could have computed the same results on a single thread and without using any asynchronous computations. However, this would be too slow, and thus we are trying to speed it up by interleaving the wait times and optimizing thread usage.

Acyclic dataflow = monadic stream

A stream that carries values of type `T` is naturally represented by a parameterized container type `Stream[T]`. Arbitrary acyclic dataflow can be implemented only if `Stream[T]` is a monadic functor: in particular, processing a single chunk of type `T` could yield another `Stream[T]` as a result. Accordingly, most streaming frameworks provide a `flatMap` operation for streams.

A monadic functor is strictly more powerful than an applicative functor, and accordingly the user has more power in implementing the processing pipeline, in which the next steps can depend in arbitrary ways on other steps and on previous data chunks in the stream. However, a monadic computation is difficult to parallelize automatically. Therefore, it is the user who now needs to specify which steps of the pipeline should be parallelized, and which steps should be separated by an asynchronous "boundary" from other steps.

Level 3: Cyclic dataflow

The class of problems I call "general dataflow" is very similar to "acyclic dataflow", except for removing the limitation that the dataflow graph should be acyclic.

The main task of general dataflow remains the same - to process data that comes as a stream, chunk after chunk. However, now we allow any step of the processing pipeline to use a "downstream" step as *input*, thus creating a loop in the dataflow graph. This loop, of course, must cross an asynchronous boundary somewhere, or else we will have an actual, synchronous infinite loop in the program. An "asynchronous infinite loop" means that the output of some downstream processing step will be *later* (asynchronously) fed into the input of some upstream step.

A typical program that requires a dataflow graph with an asynchronous loop is an event-driven graphical user interface (GUI) in the FRP paradigm. Consider, for example, an interactive Excel table with auto-updating cells will have to recompute a number of cells depending on user input events. User input events will depend on what is shown on the screen at a given time; and the contents of the screen depends on data in the cells.

This mutual dependency creates a loop in the dataflow graph: First, the user creates an input event, sending a chunk of data to the Excel computation engine. Second, the engine consults the table cells stored in memory and updates the values in the cells. Third, the updated values are shown on the screen, which allows the user to create further input events that will depend on the new contents of the cells.

The loop in the graph is asynchronous because the user creates new input events *at a later time* than the cells are updated on the screen.

This class of problems can be solved by functional reactive programming (FRP) frameworks, Akka Streams, and some other asynchronous streaming systems.

Despite the fact that the general dataflow is strictly more powerful than the acyclic dataflow, the entire computation is *still* possible to perform synchronously on a single thread without any concurrency.

Cyclic dataflow = recursive monadic stream

To formalize the difference between general and acyclic dataflow, we note that a loop in the dataflow graph is equivalent to a recursive definition of an asynchronous stream. In other words, we need the `Stream[T]` type to be a monad with a `monadFix` operation.

For instance, in a typical GUI application implemented in the FRP paradigm, one defines three streams: `Stream[Model]`, `Stream[View]`, and `Stream[Input]`. The following table illustrates the meaning of these types and the way the three streams depend on each other.

`Stream[T]`	A value of type `T` represents:	Stream depends on:
`Stream[Model]`	the data model of the application at a given time	`Stream[Input]`
`Stream[View]`	all windows and UI elements shown on the screen at a given time	`Stream[Model]`
`Stream[Input]`	any of the possible input events that the user might create	`Stream[View]`

- The model stream depends of the input stream because some input events will update the model.
- The view stream is a function of the model stream because the view shows data from the model.
- The input stream depends on the view stream because the `View` value determines which control elements are visible, and thus determines what input events the user can create while that view is shown on the screen.

Note that the last dependency is *asynchronous* because the user can create input events only *after* the view is shown. Therefore, the three streams are defined mutually recursively, and the stream graph contains an asynchronous loop.

The streaming frameworks that do not support a recursive definition of streams fall into the acyclic dataflow class.

Level 4: General concurrency

Finally, we consider the most general concurrency problem, where we need to manage many computation threads running concurrently in unknown order and interacting in arbitrary ways. For instance, one thread may start another, then at some point stop and wait until the other thread computes a certain result, and then examine that result to decide whether to continue its own computation, to wait further, or to create new computation threads.

The main difficulty here is to ensure that different threads are synchronized in the desired manner.

Frameworks such as Akka Actors, Go coroutines/channels, and the Java concurrency primitives (`Thread` , `wait/notify` , `synchronized`) are all Level 4 concurrency frameworks. The chemical machine (known in the academic world as "join calculus") is also a Level 4 framework.

A typical program that requires this level of concurrency is an operating system where many running processes can synchronize and communicate with each other in arbitrary ways.

(It seems to me that the "dining philosophers" problem is also an example of a concurrency task that cannot be implemented by any concurrency framework other than a Level 4 framework. However, I do not know how to prove that this is so.)

Why is Level 4 higher than Level 3

How do we know that Level 4 is strictly more powerful than Level 3?

I can give the following argument. Concurrency at Level 3 (and below) can be simulated on a single thread (although inefficiently). In other words, any program at Level 3 or below will give the same results when run on multiple threads and on a single thread.

If we find an example of a program that cannot be implemented on a single thread, it will follow that Level 4 is strictly more powerful than Level 3. To obtain such an example, consider the following situation:

Two objects, A and B, have methods `A.run()` and `B.run()` . Calling `run()` will start some computations that either return a value or go into an infinite loop, never returning a result. It is known that, when we call `A.run()` and `B.run()` concurrently, *at most one* of A and B can go into an infinite loop (but it could be a different process every time). We need to implement a function `firstResult(A, B)` that will run processes A and B concurrently and wait for the value returned by whichever process finishes first. The function `firstResult(A, B)` needs to return that value.

Now, we claim that this task cannot be simulated on a single thread, because no program running on a single thread can decide correctly which of the two processes returns first. Here is why: Regardless of how we implement `firstResult()`, a single-threaded program will have to call either `A.run()` or `B.run()` on that single thread. Sometimes, that call will go into an infinite loop, and then the single thread will be infinitely blocked. In that case, our program will never finish computing `firstResult()`.

So, if implemented on a single thread, `firstResult()` will sometimes fail to return a value; in other words, it is a *partial function*. However, if we are allowed to use many threads, we can implement `firstResult()` as a *total function*, always returning a value. To do that, we simply run the processes A and B simultaneously on two different threads, and we are guaranteed that at least one of the processes will return a result.

Are there other levels?

How can we be sure that there are no other levels of expressive power in concurrency?

Some assurance comes from the mathematical description of these levels:

- We need at least an applicative functor to be able to parallelize computations. This is Level 1.
- Adding `flatMap` to an applicative functor makes it into a monad, and we don't know any intermediate-power functors. Monadic streams is Level 2.
- Adding recursion raises Level 2 to Level 3. There doesn't seem to be anything in between "non-recursive" and "recursive" powers.
- Level 4 supports arbitrary concurrency. In computer science, several concurrency formalisms have been developed (Petri nets, CSP, pi-calculus, Actor model, join calculus), which are all equivalent in power to each other. I do not know of a concurrency language that is strictly less powerful than Level 4 but more powerful than Level 3.

It is possible to take a level 3 framework and add a single feature that goes beyond the expressive power of Level 3. For instance, we can add `firstResult(A,B)` as a primitive to a streaming framework. However, this single feature might not be sufficient to implement other Level 4 tasks.

Similarly, adding a primitive for starting a new thread to a Level 1 framework will enable users to perform certain tasks but not others.

I conjecture that the result of adding a single high-level feature to a lower-level framework will be a new framework that is hard to use because some concurrency tasks cannot be naturally expressed in it while others can. Users will have to work around these deficiencies

or constantly ask for new features to be added to the framework, and the design of the program will become difficult to understand.

Which level to use?

It is best to use the level of concurrency that is no higher than what is required to solve the task at hand.

For a data-parallel computation, Spark (Level 1) is the right choice while Akka Streaming (Level 3) is an overkill.

For a streaming pipeline for data processing, Scala standard streams (Level 2) are already adequate, although FS2 or Akka Streaming (Level 3) will offer more flexibility while not over-complicating the user code. Akka Actors will be an overkill, not adding much value to the application, although you will certainly be able to implement a streaming pipeline using raw actors.

For implementing a GUI, a good recursive FRP (Level 3) framework is a must. Raw Akka actors are not necessary but could be used instead of FRP.

If your task is to *implement* an alternative to Spark (Level 1) or to Scala streams (Level 2), you need a higher-powered framework, and possibly a Level 4 if you need fine-grained control over threads.

Choosing the concurrency stack

by Sergei Winitzki

Published on January 27, 2017 on LinkedIn

It has become a truism that modern software needs to be concurrent. The software engineering pundits kept telling us for the last 20 years that concurrency is the only way to process the ever-increasing data volumes, because computer chips will not get any faster.

The pundits didn't tell us *how* we are to write concurrent software. As most software engineers know, writing concurrent programs is notoriously difficult. When several computations run on different threads at once, the behavior of the program often depends sensitively on the timings of the individual computations. For this reason, a concurrent program runs slightly differently *every time* it is run, and developers can never be sure to have done "enough" debugging and testing. Books such as "Concurrent Programming in Java" describe many tricks for managing and synchronizing threads, to help developers avoid the dreaded quagmire of bugs and race conditions that they have come to expect from multithreaded programming.

Early on, Google realized the need to tackle this problem. Their solution was to develop the "map/reduce" paradigm — a much more declarative (although limited) approach to concurrent computation. It is perhaps fair to say that Google could not have succeeded in processing their humongous data sets without "map/reduce". If Google programmed their massively concurrent processing pipelines the old way, — with Java threads and semaphores, — there wouldn't have been enough engineer-hours to debug all the race conditions.

Google was followed by other Big Data companies such as Netflix and Twitter. These companies needed to develop tools for different kinds of large-volume data processing. As a result, software engineers today have at their disposal a slew of specialized concurrency frameworks: Hadoop, Spark, Flink, Kafka, Heron, Akka, just to name a few. Modern programming languages also offer high-level concurrency facilities such as parallel collections, Futures / Promises, and Async / Await. Still other options are to use the Erlang language that implements the Actor Model, or Google's Go language that incorporates coroutines and CSP channels as basic concurrency primitives. Software architects today need to make an informed choice of the concurrency stack suitable for each particular application.

One way of making sense of so many seemingly unrelated concurrency concepts is to ask what classes of problems are being solved. I think there are *four* distinct levels of complexity in concurrent programming. I call these levels “parallel data”, “acyclic streaming”, “cyclic streaming”, and “general concurrency”. As we will see, each level is a strict subset of the next. Let me explain each of them in turn.

Level 1: Parallel data

A typical data-parallel task is to produce a table of word counts in 10,000 different text files. Each text file can be processed independently of all others, which is naturally parallelized.

This class of problems is solved by "map/reduce"-like technologies such as Scala’s parallel collections, Hadoop, and Spark. The programmer manipulates the parallel data using operations such as "map", "reduce", and "filter", as if the entire data were a single array. The framework will transparently split the computation between different CPUs and/or different computers on a cluster.

Level 2: Acyclic streaming

The main task here is to process a large volume of data that is organized as one or more data streams. Each element of a stream is a small chunk of data that can be processed independently from other chunks.

In the acyclic streaming pipeline, the processing of one chunk can depend in an arbitrary way on the results of processing *previous* chunks. Because of this dependency, the "map/reduce" paradigm is unable to solve this class of problems.

The data stream can be visualized as an acyclic graph that starts upstream at "source" nodes and flows downstream, finally ending at "sink" nodes. Intermediate nodes of the graph represent processing steps that transform the chunks of data in some way. No loops in the graph are allowed — the data flows strictly in the downstream direction, although streams can fork and join.

The streaming architecture gives data engineers a lot of flexibility in scaling and optimizing the performance of the pipeline. To achieve the maximum throughput, programmers can monitor the data flow and find the processing nodes that present a performance bottleneck. At any step, the data flow can be made *asynchronous*: one step can perform its computation and deliver a result to the next step, even though that step might be still busy with its own processing. As another performance optimization, the slower processing steps could be configured to run in parallel on several data chunks at once, while other steps still run sequentially on each chunk.

A typical use case of acyclic streaming is to implement a high-throughput asynchronous Web server. When implemented with asynchronous processing steps, the server can start responding to the next request long before the previous request is answered.

Streaming frameworks include promises / futures, async / await, Akka Streaming, Flink, and the various functional reactive programming (FRP) frameworks such as reactivex.io.

Level 3: Cyclic streaming

This class of problems is very similar to acyclic streaming, except for removing the limitation that the data flow graph be acyclic.

The main task of general streaming remains the same — to process data that comes as a stream, chunk after chunk. However, now we allow any step of the processing pipeline to be used as input by a later upstream step. This creates an asynchronous loop in the data flow graph.

A typical program that requires cyclic streaming is an event-driven graphical user interface (GUI), when implemented in the FRP paradigm. Consider, for example, an interactive Excel table with auto-updating cells. The program will have to recompute a number of cells depending on user input events. User input events will depend on what is shown on the screen previously; and the contents of the screen depends on data in the cells. This mutually recursive dependency creates a loop in the data flow graph. The loop in the graph is asynchronous because the user creates new input events *later* than cells are updated on the screen.

This class of problems can be solved by functional reactive programming (FRP) frameworks such as the Elm language, Haskell's Reflex library, Akka Streaming, and some other asynchronous streaming systems.

Cyclic streaming is usually not necessary for pure data processing pipelines because they do not involve interacting with users or other processes in real time, and so there are no asynchronous loops in the data flow.

Level 4: General concurrency

Finally, consider the most general concurrency problem: to manage many computation threads running concurrently in unknown order and interacting in arbitrary ways. For instance, one thread may start another, then stop and wait until the other thread computes a certain result, and then examine that result to decide whether to continue its own computation, to wait further, or to create new computation threads.

The main difficulty here is to ensure that different threads are synchronized in the desired manner.

Frameworks such as Akka Actors, Go coroutines/channels, and the Java concurrency primitives (`Thread` , `wait` / `notify` , `synchronized`) are all Level 4 concurrency systems. Recently I have published a new open-source implementation of the abstract chemical machine (known in the academic world as "Join Calculus"), which is also a Level 4 concurrency system.

A typical program that requires this level of concurrency is an operating system where many running processes can synchronize and communicate with each other in arbitrary ways. Processes can also monitor and start or terminate other processes. This is the crucial piece of functionality that no streaming framework can provide.

Which concurrency stack to use?

As I have shown, each concurrency level is a strict subset of the next one in terms of expressive power. (I have omitted the mathematical details; for instance, in the language of type theory, the first three levels can be characterized formally as applicative, monadic, and recursive monadic functors respectively.)

Clearly, the programmer should use the level of concurrency no higher than what is required to solve the task at hand. Experience shows that when unnecessarily higher-power features are available, developers *will* use them and the code *will* become difficult to manage. It is best to limit the possibilities up front, once it is established that a certain concurrency framework is adequate for the task.

For a batch data processing pipeline, data-parallel frameworks such as Spark (Level 1) are the right choice — while Akka Streaming (Level 3) is an overkill.

For realtime data processing, a Level 2 streaming framework is usually adequate. The simplest prototype solution can use plain *Future*'s. Akka Streaming (Level 3) will offer more deployment flexibility while not significantly over-complicating the user code. However, using Akka Actors (Level 4) or Go channels (Level 4) is an overkill for that application. It is certainly possible to implement a streaming pipeline using raw actors — that's what Akka Streaming does under the hood — but the program will become unnecessarily complicated, and the maintenance of the code base will become difficult.

For implementing an event-driven concurrent GUI, a good FRP (Level 3) framework is a must. Raw Akka actors (Level 4) are not necessary for that, and while they could be used instead of FRP, it is not clear that the advantage of a more visual program design will offset the risk of using Level 4 complexity.

If your task is to *implement an alternative* to Spark (Level 1) or to Scala streams (Level 2), you need a higher-powered framework, and possibly a Level 4 if you need fine-grained control over threads. Implementing an operating system certainly requires Level 4.

Conclusion

When choosing the concurrency framework for a given task, the first step for a software architect is to classify the complexity of the task according to the four levels I outlined. This classification is broad and glosses over features of particular frameworks that could be a deal breaker. (Does Spark support file encryption? Does Kafka handle dynamic network configurations?) However, these issues need to be considered *after* determining the complexity level of the task at hand.

Concurrency stacks compete with each other and often add features that go beyond their intended complexity levels. For instance, a Level 2 framework may add certain process monitoring features that, strictly speaking, belong to Level 4. Developers need to be aware of this and exercise discipline to avoid the "paradigm leakage". If Level 2 is sufficient for implementing our application, we should not use features that belong to Level 3 or Level 4 paradigms, even if our chosen framework provides them.

Choosing the right concurrency stack *and* following the corresponding paradigm is the key to making concurrency practical.

Patterns of concurrency

To get more intuition about programming the chemical machine, let us now implement a number of simple concurrent programs. These programs are somewhat abstract and are chosen to illustrate various patterns of concurrency that are found in real software.

Allen B. Downey's *The little book of semaphores* lists many concurrency "puzzles" that he solves in Python using semaphores. In this and following chapter, we will use the chemical machine to solve those concurrency puzzles as well as some other problems.

Our approach will be deductive: we start with the problem and reason logically about the molecules and reactions that are necessary to solve it. Eventually we deduce the required chemistry and implement it declaratively in `Chymyst`.

Waiting forever

Suppose we want to implement a function `wait_forever()` that blocks indefinitely, never returning.

The chemical machine can block something indefinitely only when we emit a blocking molecule whose consuming reaction never starts. We can prevent a reaction from starting only if some input molecule for that reaction is not present in the soup. Therefore we wil make a blocking molecule `waiting_for` that reacts with another, non-blocking molecule `godot`; but `godot` never appears.

We also need to make sure that the molecule `godot()` is never emitted into the soup. To achieve this, we will declare `godot` locally within the scope of `wait_forever()`, where we *emit nothing* into the soup.

```
def wait_forever(): B[Unit, Unit] = {
  val godot = m[Unit]
  val waiting_for = b[Unit, Unit]

  site ( go { case  waiting_for(_, r) + godot(_) ⇒ r() } )

  // We do not emit `godot` here, which is the key to preventing this reaction from st
arting.

  waiting_for // Return the emitter.
}
```

The function `wait_forever()` will create and return a new blocking emitter that, when called, will block forever, never returning any value. Here is example usage:

```
val never = wait_forever() // Declare a new blocking emitter of type B[Unit, Unit].

never.timeout()(1 second) // this will time out in 1 seconds
never() // this will never return
```

Background jobs

A basic concurrency pattern is to start a long background job and to get notified when the job is finished.

It is easy to come up with a suitable chemistry. The reaction needs no data to start, and the computation can be inserted directly into the reaction body. So we define a reaction with a single non-blocking input molecule. The reaction will consume the molecule, do the long computation, and then emit a `finished(...)` molecule that carries the result value of the computation.

A convenient implementation is to define a function that will return an emitter that starts the job.

```
/**
* Prepare reactions that will run a closure
* and emit a result upon its completion.
*
* @tparam R The type of result value
* @param closure  The closure to be run
* @param finished A previously bound non-blocking molecule
*                 to be emitted when the computation is done
* @return A new non-blocking molecule that will start the job
*/
def submitJob[R](closure: () ⇒ R, finished: M[R]): M[R] = {
  val startJobMolecule = m[Unit] // Declare a new emitter.

  site (
    go { case startJobMolecule(_) ⇒
      val result = closure()
      finished(result)
    }
  )

  startJobMolecule // Return the new emitter.
}
```

The `finished` molecule should be bound to another reaction site.

Another implementation of the same idea will put the `finished` emitter into the molecule value, together with the closure that needs to be run.

However, we lose some polymorphism since Scala values cannot be parameterized by a type. The `startJobMolecule` cannot have type parameters and so has to carry values of type `Any`:

```
val startJobMolecule = new M[() ⇒ Any, M[Any]]

site (
  go {
    case startJobMolecule(closure, finished) ⇒
      val result = closure()
      finished(result)
  }
)
```

A solution to this difficulty is to create a method that is parameterized by type and returns a `startJobMolecule`:

```
def makeStartJobMolecule[R]: M[(() ⇒ R, M[R])] = {
  val startJobMolecule = m[(() ⇒ R, M[R])]

  site (
    go {
      case startJobMolecule(closure, finished) ⇒
        val result = closure()
        finished(result)
   }
  )
  startJobMolecule
}
```

Waiting until `n` jobs are finished: non-blocking calls

A frequently used pattern is to start `n` concurrent jobs and wait until all of them are finished.

Suppose that we have started `n` jobs and each job, when done, will emit a *non-blocking* molecule `done()`. We would like to implement a blocking molecule `all_done()` that will block until `n` molecules `done()` are emitted.

To begin reasoning about the necessary molecules and reactions, consider that `done()` must react with some other molecule that keeps track of how many `done()` molecules remain to be seen. Call this other molecule `remaining(k)` where `k` is an integer value

showing how many `done()` molecules were already seen. The reaction should have the form `done() + remaining(k) ⇒ ...`, and it is clear that the reaction should consume `done()`, otherwise the reaction will start with it again. So the reaction will look like this:

```
val done = m[Unit]
val remaining = m[Int]

site(
  go { done(_) + remaining(k) ⇒ remaining(k - 1) }
)
remaining(n) // Emit the molecule with value `n`,
// which is the initial number of remaining `done()` molecules.
```

Now, it remains to implement waiting until all is done. The blocking molecule `all_done()` should start its reaction only when we have `remaining(0)` in the soup. Therefore, the reaction is

```
val all_done = b[Unit,Unit]

go { all_done(_, reply) + remaining(0) ⇒ reply() }
```

Since this reaction consumes `remaining()`, it should be declared at the same reaction site as the `{ done + remaining ⇒ ... }` reaction.

The complete code is

```
val done = m[Unit]
val remaining = m[Int]
val all_done = b[Unit,Unit]

site(
  go { done(_) + remaining(k) if k > 0 ⇒ remaining(k - 1) }, // Adding a guard to be s
afe.
  go { all_done(_, reply) + remaining(0) ⇒ reply() }
)
remaining(n) // Emit the molecule with value `n`,
// which is the initial number of remaining `done()` molecules.
```

Adding a guard (`if k > 0`) will prevent the first reaction from running once we consume the expected number of `done()` molecules. This allows the user to emit more `done()` molecules than `n`, without disrupting the logic.

Example usage

How would we use this code? Suppose we have `n` copies of a reaction whose completion we need to wait for. At the end of that reaction, we will now emit a `done()` molecule. After emitting the input molecules for these reactions to start, we call `all_done()` and wait for the completion of all jobs:

```
val begin = m[Int]

site(
  go { begin(x) ⇒ long_computation(x); done() )}
)
val n = 10000
(1 to n).foreach(begin) // Emit begin(1), begin(2), ..., begin(10000) now.

all_done() // This will block until all reactions are finished.
```

Refactoring into a function

Consider what is required in order to refactor this chemistry into a reusable function such as `make_all_done()` that would create the `all_done()` molecule for us.

The function `make_all_done()` will need to declare a new reaction site. The `done()` molecule is an input molecule at the new reaction site. Therefore, this molecule cannot be already defined before we perform the call to `make_all_done()`.

We see that the result of the call `make_all_done()` must be the creation of *two* new molecules: a new `done()` molecule and a new `all_done()` molecule.

The user will then need to make sure that every job emits the `done()` molecule at the end of the job, and arrange for all the jobs to start. When it becomes necessary to wait until the completion of all jobs, the user code will simply emit the `all_done()` blocking molecule and wait until it returns.

Refactoring an inline piece of chemistry into a reusable function is generally done in two steps:

- within the function scope, declare the same molecules and reactions as in the previous inline code;
- at the end of the function body, return just the new molecule emitters that the user will need to call, but no other molecule emitters.

Here is the result of this refactoring for our previous code:

```
def make_all_done(n: Int): (M[Unit], B[Unit, Unit]) = {
  val done = m[Unit]
  val remaining = m[Int]
  val all_done = b[Unit,Unit]

  site(
    go { done(_) + remaining(k) if k > 0 ⇒ remaining(k - 1) }, // Adding a guard to be
 safe.
    go { all_done(_, reply) + remaining(0) ⇒ reply() }
  )
  remaining(n)

  (done, all_done)
}
```

Let us now use the method `make_all_done()` to simplify the example usage code we had in the previous section:

```
val begin = m[Int]
val n = 10000

val (done, all_done) = make_all_done(n)

site(
  go { begin(x) ⇒ long_computation(x); done() )}
)

(1 to n).foreach(begin) // Emit begin(1), begin(2), ..., begin(10000) now.

all_done() // This will block until all reactions are finished.
```

Waiting until `n` jobs are finished: blocking calls

A variation on the same theme is to detect when `n` jobs are finished, given that each job will emit a *blocking* molecule `done()` at the end.

In the previous section, we encapsulated the functionality of waiting for `n` non-blocking molecules into a function `make_all_done()`. Let us take the code we just saw for `make_all_done()` and try to modify it for the case when `done()` is a blocking molecule. The key reaction

```
go { done(_) + remaining(k) if k > 0 ⇒ remaining(k - 1) }
```

now needs to be modified because we must reply to the `done()` molecule at some point in that reaction:

```
go { done(_, r) + remaining(k) if k > 0 ⇒ r() + remaining(k - 1) }
```

In this way, we unblock whatever final cleanup the job needs to do after it signals `done()`. All other code in `make_all_done()` remains unchanged.

Control over a shared resource (mutex, multiplex)

Single access

Suppose we have an application with many concurrent processes and a shared resource *R* (say, a database server) that should be only used by one process at a time. Let us assume that there is a certain function `doWork()` that will use the resource *R* when called. Our goal is to make sure that `doWork()` is only called by at most one concurrent process at any time. While `doWork()` is being evaluated, we consider that the resource *R* is not available. If `doWork()` is already being called by one process, all other processes trying to call `doWork()` should be blocked until the first `doWork()` is finished and the resource *R* is again available.

How would we solve this problem using the chemical machine? Since our only way to control concurrency is by manipulating molecules, we need to organize the chemistry such that `doWork()` is only called when certain molecules are available. In other words, `doWork()` must be called by a *reaction* that consumes certain molecules whose presence or absence we will control. The reaction must be of the form `case [some molecules] ⇒ ... doWork() ...`. Let us call it the "worker reaction". Our code must be such that the only way to call `doWork()` is by starting this reaction.

Processes that need to call `doWork()` will therefore need to emit a certain molecule that the worker reaction consumes. Let us call that molecule `request()`. The `request()` molecule must be a *blocking* molecule because `doWork()` should be able to block the caller when the resource *R* is not available.

If the `request()` molecule is the only input molecule of the worker reaction, we will be unable to prevent the reaction from starting whenever some process emits `request()`. Therefore, we need a *second* input molecule in the worker reaction. Let us call that molecule `access()`. The worker reaction will then look like this:

```
go { case access(_) + request(_, r) ⇒ ... doWork() ... }
```

Suppose some process emits a `request` molecule, and suppose this is the only process that does so at the moment. We should then allow the worker reaction to start and to complete `doWork()`. Therefore, `access()` must be present at the reaction site: we should have emitted it beforehand.

While `doWork()` is evaluated, the `access()` molecule is absent, so no other copy of the worker reaction can start. This is precisely the exclusion behavior we need.

When the `doWork()` function finishes, we need to unblock the calling process by replying to the `request()` molecule. Perhaps `doWork()` will return a result value: in that case, we should pass this value as the reply value. We should also emit the `access()` molecule back into the soup, so that another process will be able to run `doWork()`.

After these considerations, the worker reaction becomes

```
site (
  go { case access(_) + request(_, reply) ⇒ reply(doWork()) + access() }
)
access() // Emit just one copy of `access`.
```

As long as we emit just one copy of the `access()` molecule, and as long as `doWork()` is not used elsewhere in the code, we will guarantee that at most one process will call `doWork()` at any time.

Multiple access

What if we need to relax the requirement of single access for the resource *R*? Suppose that now, at most `n` concurrent processes should be able to call `doWork()`.

This formulation of the shared resource problem will describe, for instance, the situation where a connection pool with a fixed number of connections should be used to access a database server.

The effect of allowing `n` simultaneous reactions to access the resource *R* can be achieved simply by initially emitting `n` copies of the `access()` molecule. The worker reaction remains unchanged. This is how easy it is to program the chemical machine!

Refactoring into a function

The following code defines a convenience function that wraps `doWork()` and provides single-access or multiple-access restrictions. This illustrates how we can easily and safely package new chemistry into a reusable function.

```
def wrapWithAccess[T](allowed: Int, doWork: () ⇒ T): () ⇒ T = {
  val access = m[Unit]
  val request = b[Unit, T]

  site (
    go { case access(_) + request(_, reply) ⇒ reply(doWork()) + access() }
  )

  (1 to n).foreach(access) // Emit `n` copies of `access`.
  val result: () ⇒ T = () ⇒ request()
  result
}
// Example usage:
val doWorkWithTwoAccesses = wrapWithAccess(2, () ⇒ println("do work"))
// Now `doWork()` will be called by at most 2 processes at a time.

// ... start a new process, in which:
doWorkWithTwoAccesses()
```

Token-based access

It is often needed to regulate access to resource *R* by tokens that carry authentication or other information. We will now suppose that `doWork()` requires a token, and that we only have a limited set of tokens. Therefore, we need to make sure that

- each process that wants to call `doWork()` receives a token from the token set;
- if all tokens are taken, calls to `doWork()` by other processes are blocked until more tokens become available.

To implement this, we can use the `n`-access restriction on the worker reaction. We just need to make sure that every worker reaction receives a token that enables it to `doWork()`, and that it returns the token when the access is no longer needed.

Since the worker reaction already consumes the `access()` molecule, it is clear that we can easily pass the token as the value carried by `access()`. We just need to change the type of `access` from `M[Unit]` to `M[TokenType]`, where `TokenType` is the type of the value that we need (which could be a string, a thread, a resource handle, etc.).

Here is the modified code:

```
def wrapWithAccessTokens[T, TokenType](tokens: Set[TokenType], doWork: TokenType ⇒ T):
 () ⇒ T = {
  val access = m[TokenType]
  val request = b[Unit, T]

  site (
    go { case access(token) + request(_, reply) ⇒ reply(doWork(token)); access(token)
}
  )

  tokens.foreach(access) // Emit `tokens.size` copies of `access(...)`, putting a toke
n on each molecule.
  val result: () ⇒ T = () ⇒ request()
  result
}
// Example usage:
val doWorkWithTwoAccesses = wrapWithAccessTokens(Set("token1", "token2"), t ⇒ println(
s"do work with token $t"))
// Now `doWork()` will be called by at most 2 processes at a time.

// ... start a new process, in which:
doWorkWithTwoAccesses()
```

When concurrent processes emit `request()` molecules, only at most `n` processes will actually do work on the resource at any one time. Each process will be assigned a token out of the available set of tokens.

Concurrent critical sections, or `Object.synchronized`

A "critical section" is a portion of code that cannot be safely called from different processes at the same time. We will now implement the following requirements:

- The calls `beginCritical()` and `endCritical()` can be made in any reaction, in this order.
- The code between these two calls can be executed only by one reaction at a time, among all reactions that call these functions.
- A reaction that calls `beginCritical()` will be blocked if another reaction already called `beginCritical()` but did not yet call `endCritical()`, and will be unblocked only when that other reaction calls `endCritical`.

This functionality is similar to `Object.synchronized`, which provides exclusive synchronized access within otherwise reentrant code. In our case, the critical section is delimited by two function calls, `beginCritical()` and `endCritical()`, that can be made at any two points in

the code — including `if` expressions or inside closures, which is impossible with `Object.synchronized`.

The `Object.synchronized` construction identifies the synchronized blocks by an `Object` reference: different objects will be responsible for synchronizing different blocks of reentrant code. What we would like to allow is *any* code anywhere to contain any number of critical sections identified by the same `beginCritical()` call.

How can we implement this functionality using the chemical machine? Since the only way to communicate between reactions is to emit molecules, `beginCritical()` and `endCritical()` must be molecules. Clearly, `beginCritical` must be blocking while `endCritical` does not have to be; so let us make `endCritical` a non-blocking molecule.

What should happen when a process emits `beginCritical()`? We must enforce a single-user exclusive access to the critical section. In other words, only one reaction that consumes `beginCritical()` should be running at any one time, even if several `beginCritical` molecules are available. Therefore, this reaction must also require another input molecule, say `access()`, of which we will only have a single copy emitted into the soup:

```
val beginCritical = b[Unit, Unit]
val access = m[Unit]

site(
  go { case beginCritical(_, reply) + access(_) ⇒ ???; reply(); ??? }
)
access() // Emit only one copy.
```

Just as in the case of single-access resource, we can guarantee that `beginCritical()` will block if called more than once. All we need to do is insure that there is initially a single copy of `access()` in the soup, and that no further copies of `access()` are ever emitted.

Another requirement is that emitting `endCritical()` must end whatever `beginCritical()` began, but only if `endCritical()` is emitted by the *same reaction*. If a different reaction emits `endCritical()`, no access to the critical section should be granted. Somehow, the `endCritical()` molecules must be distinct when emitted by different reactions. However, `beginCritical()` must be the same for all reactions, or else there can't be any contention between them.

Additionally, we would like it to be impossible for any reaction to call `endCritical()` without first calling `beginCritical()`.

One way of achieving this is to make `beginCritical()` return a value that is required for us to call `endCritical()` later. The simplest such value is the emitter `endCritical` itself. So let us make `beginCritical()`, which is a blocking call, return the `endCritical` emitter.

To achieve the uniqueness of `endCritical`, let us implement the call to `beginCritical()` in such a way that it creates each time a new, unique emitter for `endCritical()`, and returns that emitter:

```
val beginCritical = b[Unit, M[Unit]]
val access = m[Unit]

site (
  go { case beginCritical(_, reply) + access(_) ⇒
    val endCritical = m[Unit] // Declare a new emitter, unique for this call to `begin
Critical`.
    site (
      go { case endCritical(_) ⇒ ??? }
    )
    reply(endCritical) // beginCritical() returns the new emitter.
  }
)
access() // Emit only one copy.
```

Since `endCritical()` and its chemistry are defined within the local scope of the reaction that consumes `beginCritical()`, each such scope will create a *chemically unique* new molecule that no other reactions can emit. This will guarantee that reactions cannot end the critical section for other reactions.

Also, since the `endCritical` emitter is created by `beginCritical()`, we cannot possibly call `endCritical()` before calling `beginCritical()`! So far, so good.

It remains to make `endCritical()` do something useful when called. The obvious thing is to make it emit `access()`. The presence of `access()` in the soup will restore the ability of other reactions to enter the critical section.

The code then looks like this:

```
val beginCritical = b[Unit, M[Unit]]
val access = m[Unit]

site (
  go { case beginCritical(_, reply) + access(_) ⇒
    val endCritical = m[Unit] // Declare a new emitter.
    site (
      go { case endCritical(_) ⇒ access() }
    )
    reply(endCritical) // beginCritical() returns the new emitter.
  }
)
access() // Emit only one copy.

// Example usage:
val endCritical = beginCritical()

???... // The code of the critical section.

endCritical() // End of the critical section.
```

This implementation works but has a drawback: the user may call `endCritical()` multiple times. This will emit multiple `access()` molecules and break the logic of the critical section functionality.

To fix this problem, let us think about how we could prevent `endCritical()` from starting its reaction after the first time it did so. The only way to prevent a reaction from starting is to omit an input molecule. Therefore, we need to introduce another molecule (say, `beganOnce`) as input into that reaction. The reaction will consume that molecule and never emit it again.

```
val beginCritical = b[Unit, M[Unit]]
val access = m[Unit]

site (
  go { case beginCritical(_, reply) + access(_) ⇒
    val endCritical = m[Unit] // Declare a new emitter.
    val beganOnce = m[Unit]
    site (
      go { case endCritical(_) + beganOnce(_) ⇒ access() }
    )
    beganOnce() // Emit only one copy.
    reply(endCritical) // beginCritical() returns the new emitter.
  }
)
access() // Emit only one copy.

// Example usage:
val endCritical = beginCritical()

???... // The code of the critical section.

endCritical() // End of the critical section.

endCritical() // This has no effect because `beganOnce()` is not available any more.
```

As before, we can easily modify this code to support multiple (but limited) concurrent entry into critical sections, or to support token-based access.

Refactoring into a function

We would like to be able to create new, unique `beginCritical()` molecules on demand, so that we could have multiple distinct critical sections in our code.

For instance, we could declare two critical sections and use them in three reactions that could run concurrently like this:

```
val beginCritical1 = newCriticalSectionMarker()
val beginCritical2 = newCriticalSectionMarker()
```

The result should be that we are able to delimit any number of critical sections that work independently of each other.

Reaction 1		Reaction 2		Reaction 3
`beginCritical1()`		...		`beginCritical2()`
`beginCritical2()`		...		...
(blocked by 3)		`beginCritical1()`		...
(starts running)		(blocked by 1)		`endCritical2()`
...		(still blocked)		`beginCritical2()`
`endCritical1()`		(starts running)		(blocked by 1)
`endCritical2()`		...		(starts running)
...		...		...
...		`endCritical1()`		`endCritical2()`

In order to package the implementation of the critical section into a function `newCriticalSectionMarker()`, we simply declare the chemistry in the local scope of that function and return the molecule emitter:

```
def newCriticalSectionMarker(): B[Unit, M[Unit]] = {

  val beginCritical = b[Unit, M[Unit]]
  val access = m[Unit]

  site (
    go { case beginCritical(_, reply) + access(_) ⇒
      val endCritical = m[Unit] // Declare a new emitter.
      val beganOnce = m[Unit]
      site (
        go { case endCritical(_) + beganOnce(_) ⇒ access() }
      )
      beganOnce() // Emit only one copy.
      reply(endCritical) // beginCritical() returns the new emitter.
    }
  )
  access() // Emit only one copy.

  beginCritical // Return the new emitter.
}

// Example usage:
val beginCritical1 = newCriticalSectionMarker()
// Now we can pass the `beginCritical1` emitter value to several reactions.
// Suppose we are in one of those reactions:
val endCritical = beginCritical1()

???... // The code of the critical section 1.

endCritical() // End of the critical section.
```

Rendezvous, or `java.concurrent.Exchanger`

The "rendezvous" problem is to implement two concurrent processes that perform some computations and wait for each other like this:

Process 1		Process 2
`val x1 =` compute something		`val x2 =` compute something
send `x1` to Process 2, wait for reply		send `x2` to Process 1, wait for reply
`val y1 =` what Process 2 computed as its `x2`		`val y2 =` what Process 1 computed as its `x1`
`val z = further_computations_1(y1)`		`val z = further_computations_2(y2)`

(This functionality is essentially that of `java.concurrent.Exchanger`.)

Let us now figure out the chemistry that will solve this problem.

The two processes must be reactions (since any computation that runs in the chemical machine is a reaction). These reactions must start by consuming some initial molecules. Let us start by defining these molecules and reactions, leaving undefined places for the next steps:

```
val begin1 = m[Unit]
val begin2 = m[Unit]

site(
  go { case begin1(_) ⇒
    val x1 = 123 // some computation
    ??? // send x1 to Process 2 somehow
    val y1 = ??? // receive value from Process 2
    val z = further_computation_1(y1)
   },
  go { case begin2(_) ⇒
    val x2 = 456 // some computation
    ??? // send x2 to Process 1 somehow
    val y2 = ??? // receive value from Process 1
    val z = further_computation_2(y2)
   }
)
begin1() + begin2() // emit both molecules to enable starting the two reactions
```

Let us now look at what happens in Process 1 after `x1` is computed. The next step at that point is to send the value `x1` to Process 2. The only way of sending data to another process is by emitting a molecule with a value. Therefore, here we must be emitting *some molecule*.

Now, either Process 1 or Process 2 is already running by this time, and so it won't help if we emit a molecule that Process 2 consumes as input. Therefore, we must emit a new molecule that neither Process 1 nor Process 2 consume as input.

Also note that each process must wait until the other process sends back its value. Therefore, the new molecule must be a blocking molecule.

Let's say that each process will emit a blocking molecule `barrier()` at the point when it must wait for the other process. The code will look like this:

```
val begin1 = m[Unit]
val begin2 = m[Unit]

val barrier = b[Unit,Unit]

site(
  go { case begin1(_) ⇒
    val x1 = 123 // some computation
    barrier(x1)
    ??? // send x1 to Process 2 somehow
    val y1 = ??? // receive value from Process 2
    val z = further_computation_1(y1)
   },
  go { case begin2(_) ⇒
    val x2 = 456 // some computation
    barrier(x2)
    ??? // send x2 to Process 1 somehow
    val y2 = ??? // receive value from Process 1
    val z = further_computation_2(y2)
   }
)
begin1() + begin2() // emit both molecules to enable starting the two reactions
```

Now we note that blocking molecules can receive reply values. Therefore, the call to `barrier1` may receive a reply value. This is exactly what we need! Let's make `barrier1` return the value that Process 2 sends, and vice versa.

```
val begin1 = m[Unit]
val begin2 = m[Unit]

val barrier = b[Int,Int]

site(
  go { case begin1(_) ⇒
    val x1 = 123 // some computation
    val y1 = barrier(x1) // receive value from Process 2
    val z = further_computation_1(y1)
   },
  go { case begin2(_) ⇒
    val x2 = 456 // some computation
    val y2 = barrier(x2) // receive value from Process 1
    val z = further_computation_2(y2)
   }
)
begin1() + begin2() // emit both molecules to enable starting the two reactions
```

At this point, the molecule `barrier()` is not yet consumed by any reactions. We now need to define some reaction that consumes these molecules.

The two processes will each emit one copy of `barrier()`. It is clear that what we need is a reaction that exchanges the values these two molecules carry. The easiest solution is to just let these two molecules react with each other. The reaction will then reply to both of them, exchanging the reply values.

```
go { case barrier(x1, reply1) + barrier(x2, reply2) ⇒ reply1(x2) + reply2(x1) }
```

The final code looks like this:

```
val begin1 = m[Unit]
val begin2 = m[Unit]

val barrier = b[Int,Int]

site(
  go { case begin1(_) ⇒
    val x1 = 123 // some computation
    val y1 = barrier(x1) // receive value from Process 2
    val z = further_computation_1(y1)
   },
  go { case begin2(_) ⇒
    val x2 = 456 // some computation
    val y2 = barrier(x2) // receive value from Process 1
    val z = further_computation_2(y2)
   },
   go { case barrier(x1, reply1) + barrier(x2, reply2) ⇒ reply1(x2) + reply2(x1) }
)
begin1() + begin2() // emit both molecules to enable starting the two reactions
```

Working test code for the rendezvous problem is in `Patterns01Spec.scala`.

Rendezvous with `n` participants

Suppose we need a rendezvous with `n` participants: there are `n` processes (where `n` is a run-time value, not known in advance), and each process would like to wait until all other processes reach the rendezvous point. After that, all `n` waiting processes become unblocked and proceed concurrently.

This is similar to the rendezvous with two participants that we just discussed. To simplify the problem, we assume that no data is exchanged — this is a pure synchronization task.

Let us try to generalize our previous implementation of the rendezvous from 2 participants to `n`. Since emitters are values, we could define `n` different emitters `begin1`, ..., `begin_n`, `barrier1`, ..., `barrier_n` and so on. We could store these emitters in an array and also

define an array of corresponding reactions.

However, we notice a problem when we try to generalize the reaction that performs the rendezvous:

```
go { case barrier(x1, reply1) + barrier(x2, reply2) ⇒ ... }
```

Generalizing this reaction straightforwardly to `n` participants would now require a reaction with `n` input molecules. However, input molecules to each reaction must be defined statically at compile time. Since `n` is a run-time parameter, we cannot define a reaction with `n` input molecules. So we cannot generalize from 2 to `n` participants in this way.

What we need is to consume all `n` blocking molecules `barrier()` and also reply to all of them after we make sure we consumed exactly `n` of them. The only way to implement chemistry that consumes `n` molecules (where `n` is a run-time parameter) is to consume one molecule at a time while counting to `n`. We need a molecule, say `counter()`, to carry the integer value that shows the number of `barrier()` molecules already consumed. Therefore, we need a reaction like this:

```
go { case barrier(_, reply) + counter(k) ⇒
  ???
  if (k + 1 < n) counter(k + 1); ???; reply()
}
```

This reaction must reply to the `barrier()` molecule, since a reply is required in any reaction that consumes a blocking molecule. It is clear that `reply()` should come last: this is the reply to the `barrier()` molecule, which should be performed only after we counted up to `n`. Until then, this reaction must be blocked.

The only way to block in the middle of a reaction is to emit a blocking molecule there. Therefore, some blocking molecule must be emitted in this reaction before replying to `barrier()`. Let us make `counter()` a blocking molecule:

```
go { case barrier(_, reply) + counter(k, replyCounter) ⇒
  ???
  if (k + 1 < n) counter(k + 1); ???; reply()
}
```

What is still missing is the reply to the `counter(k)` molecule. Now we are faced with a question: should we perform that reply before emitting `counter(k + 1)` or after? Here are the two possible reactions:

```
val reaction1 = go { case barrier(_, reply) + counter(k, replyCounter) ⇒
  replyCounter(); if (k + 1 < n) counter(k + 1); reply()
}
val reaction2 = go { case barrier(_, reply) + counter(k, replyCounter) ⇒
  if (k + 1 < n) counter(k + 1); replyCounter(); reply()
}
```

We now need to decide whether `reaction1` or `reaction2` works as required.

To resolve this question, let us visualize the molecules that we need to be present at different stages of running the program. Suppose that `n = 10` and we have already consumed all `barrier()` molecules except three. We presently have three `barrier()` molecules and one `counter(7)` molecule. (As we decided, this is a blocking molecule.)

Note that the `counter(7)` molecule was emitted by a previous reaction of the same kind,

```
go { barrier() + counter(6) ⇒ ... counter(7); ... }
```

So, this previous reaction is now blocked at the place where it emitted `counter(7)`.

Next, the reaction `{ barrier() + counter(7) ⇒ ... }` will start and consume its input molecules, leaving two `barrier()` molecules present.

Now the reaction body of `{ barrier() + counter(7) ⇒ ... }` will run and emit `counter(8)`. Then we will have the molecules `barrier(), barrier(), counter(8)` in the soup. This set of molecules will be the same whether we use `reaction1` or `reaction2`.

Suppose we used `reaction1`. We see that in the body of `reaction1` the reply is sent to `counter(7)` before emitting `counter(8)`. The reply to `counter(7)` will unblock the previous reaction,

```
{ case barrier(_, reply) + counter(6, replyCounter) ⇒ replyCounter(); counter(7); repl
y() }
```

which was blocked at the call to `counter(7)`. Now this reaction can proceed to reply to its `barrier()` using `reply()`. However, at this point it is too early to send a reply to `barrier()`, because we still have two `barrier()` molecules that we have not yet consumed!

Therefore, `reaction1` is incorrect. On the other hand, `reaction2` works correctly. It will block at each emission of `counter(k + 1)` and unblock only when `k = n - 1`. At that time, `reaction2` will reply to the `counter(n - 1)` molecule and to the `barrier()` molecule just consumed. The reply to the `counter(n - 1)` molecule will unblock the copy of `reaction2` that emitted it, which will unblock the next molecules.

In this way, the `n`-rendezvous will require all `n` reactions to wait at the "barrier" until all participants reach the barrier, and then unblock all of them.

It remains to see how the very first `barrier()` molecule can be consumed. We could emit a `counter(0)` molecule at the initial time. This molecule is blocking, so emitting it will block its emitter until the very end of the `n`-rendezvous. This may be undesirable.

To avoid that, we can introduce a special initial molecule `counterInit()` with a reaction such as

```
go { case barrier(_, reply) + counterInit(_) ⇒ counter(1); reply() }
```

This works; the complete code looks like this:

```
val barrier = b[Unit,Unit]
val counterInit = m[Unit]
val counter = b[Int,Unit]

site(
  go { case barrier(_, reply) + counterInit(_) ⇒
    // this reaction will consume the very first barrier molecule emitted
    counter(1) // one reaction has reached the rendezvous point
    reply()
  },
  go { case barrier(_, reply) + counter(k, replyCounter) ⇒
    if (k + 1 < n) counter(k + 1) // k + 1 reactions have reached the rendezvous point
    replyCounter()
    reply()
  }
)
counterInit() // This needs to be emitted initially.
```

Note that this chemistry will block `n` reactions that emit `barrier()` and another set of `n` copies of `reaction2`. In the current implementation of `Chymyst`, a blocked reaction always blocks a thread, so the `n`-rendezvous will block `2*n` threads until the rendezvous is passed. (A future implementation of `Chymyst` might be able to perform a code transformation that never blocks any threads.)

How do we use the `n`-rendezvous? Suppose we have a reaction where a certain processing step needs to wait for all other reactions to reach the same step. Then we simply emit the `barrier()` molecule at that step:

```
site (
  go { case begin(_) ⇒
    work()
    barrier() // need to wait here for other reactions
    more_work()
  }
)

(1 to 1000).foreach (begin) // start 1000 copies of this reaction
```

Refactoring into a function

As always when encapsulating some piece of chemistry into a reusable function, we first figure out what new molecule emitters are necessary for the users of the function. These new emitters will be returned by the function, while all the other chemistry will remain encapsulated within that function's local scope.

In our case, the users of the function will only need the `barrier` emitter. In fact, users should *not* have access to `counter` or `counterInit` emitters: if users emit any additional copies of these molecules, the encapsulated reactions will begin to work incorrectly.

We also note that the number `n` needs to be given in advance, before creating the reaction site for the `barrier` molecule. Therefore, we write a function with an integer argument `n`:

```
def makeRendezvous(n: Int): B[Unit, Unit] = {
  val barrier = b[Unit,Unit]
  val counterInit = m[Unit]
  val counter = b[Int,Unit]

  site(
    go { case barrier(_, reply) + counterInit(_) ⇒
      // this reaction will consume the very first barrier molecule emitted
      counter(1) // one reaction has reached the rendezvous point
      reply()
    },
    go { case barrier(_, reply) + counter(k, replyCounter) ⇒
      if (k + 1 < n) counter(k + 1) // k + 1 reactions have reached the rendezvous poi
nt
      replyCounter()
      reply()
    }
  )
  counterInit() // This needs to be emitted initially.

  barrier // Return only this emitter.
}
```

The usage example will then look like this:

```
val barrier = makeRendezvous(1000)

site (
  go { case begin(_) ⇒
    work()
    barrier() // need to wait here for other reactions
    more_work()
  }
)

(1 to 1000).foreach (begin) // start 1000 copies of this reaction
```

Reusable `n` -rendezvous

The `n` -rendezvous as implemented in the previous section has a drawback: Once `n` reactions have passed the rendezvous point, emitting more `barrier()` molecules will have no effect. If another set of `n` reactions needs to participate in an `n` -rendezvous, a new call to `makeRendezvous()` needs to be made in order to produce a whole new reaction site with a new `barrier()` molecule.

In a *reusable* `n` -rendezvous, once a set of `n` participant reactions have passed the rendezvous point, the same `barrier` molecule should automatically become ready to be used again by another set of `n` reactions. This can be seen as a batching functionality: If reactions emit a large number of `barrier()` molecules, these molecules are split into batches of size `n` . A rendezvous procedure is automatically performed for one batch after another.

Let us see how we can revise the code of `makeRendezvous()` to implement this new requirement. We need to consider what molecules will be present after one rendezvous is complete. With our present implementation, the reaction site will have no molecules present because the `counter()` molecule will be consumed at the last iteration, and `counterInit()` was consumed at the very beginning. At that point, emitting more `barrier()` molecules will therefore not start any reactions.

In order to allow starting the `n` -rendezvous again, we need to make sure that `counterInit()` is again present. Therefore, the only needed change is to emit `counterInit()` when the `n` -rendezvous is complete:

```
if (k + 1 < n) counter(k + 1) else counterInit()
```

After this single change, the `n` -rendezvous function becomes a reusable `n` -rendezvous.

Exercises

Superadmin `n`-rendezvous

Revise the `n`-rendezvous chemistry to allow one "superadmin" reaction, in addition to ordinary reactions. The "superadmin" reaction can unlock the barrier no matter how many other reactions have reached the barrier so far. In other words, the "superadmin-`n`-rendezvous" is achieved when either `n` reactions reach the barrier, or the `superadmin` reaction reaches the barrier regardless of how many other reactions have reached so far. After the superadmin unlocks the barrier, any other reactions can freely pass the barrier.

Weighted `n`-rendezvous

Revise the `n`-rendezvous chemistry to allow different "weights" for reactions waiting at the barrier. Weights are integers. The `n`-rendezvous is passed when enough reactions reach the barrier so that the sum of all their weights equals `n`.

Pair up for dance

In the 18th century Paris, there are two doors that open to the dance floor where men must pair up with women to dance. At random intervals, men and women arrive to the dancing place. Men queue up at door A, women at door B.

The first man waiting at door A and the first woman waiting at door B will then form a pair and go off to dance. If a man is first in the queue at door A but no woman is waiting at door B (that is, the other queue is empty), the man needs to wait until a woman arrives and goes off to dance with her. Similarly, when a woman is first in the queue at door B and no man is waiting, she needs to wait for the next man to arrive, and then go off to dance with him.

(This problem is sometimes called "Leaders and Followers", but this name is misleading since the roles of the two dance partners are completely symmetric.)

Let us implement a simulation of this problem in the chemical machine.

The problem is about controlling the starting of a reaction that represents the "dancing" computation. The reaction can start when a man and a woman are present. It is clear that we can simulate this via two molecules, `man` and `woman`, whose presence is required to start the reaction.

```
go { case man(_) + woman(_) ⇒ beginDancing() }
```

To simplify this example, let us assume that some other reactions will randomly emit `man()` and `woman()` molecules.

The problem with the above reaction is that it does not necessarily respect the linear order of molecules in the queue. If several `man()` and `woman()` molecules are emitted quickly enough, they will be paired up in random order, rather than in the order of emission. Also, nothing prevents several pairs to begin dancing at once, regardless of the dancer's positions in the queues.

How can we enforce the order of arrival on the pairs?

The only way to do that is to label each `man` and `woman` molecule with an integer that represents their position in the queue. However, the external reactions that emit `man` and `woman` will not know about our ordering requirements. Therefore, to ensure that the labels are given out consistently, we need our own reactions that will assign position labels.

Let us define new molecules, `manL` representing "man with label" and `womanL` for "woman with label". The dancing reaction will become something like this,

```
val manL = m[Int]
val womanL = m[Int]
go { case manL(m) + womanL(w) if m == w ⇒ beginDancing() }
```

The last positions in the men's and women's queues should be maintained and updated as new dancers arrive. Since the only way of keeping state is by putting data on molecules, we need new molecules that hold the state of the queue. Let us call these molecules `queueMen` and `queueWomen`. We can then define reactions that produce `manL` and `womanL` with correct position labels:

```
val man = m[Unit]
val manL = m[Int]
val queueMen = m[Int]
val woman = m[Unit]
val womanL = m[Int]
val queueWomen = m[Int]
val beginDancing = m[Unit]

site(
  go { case man(_) + queueMen(n) ⇒ manL(n) + queueMen(n+1) },
  go { case woman(_) + queueWomen(n) ⇒ womanL(n) + queueWomen(n+1) }
)
```

The result of this chemistry is that a number of `manL` and `womanL` molecules may accumulate at the reaction site, each carrying their position label. We now need to make sure they start dancing in the order of their position.

For instance, we could have molecules `manL(0)`, `manL(1)`, `manL(2)`, `womanL(0)`, `womanL(1)` at the reaction site. In that case, we should first let `manL(0)` and `womanL(0)` pair up and begin dancing, and only when they have done so, we may pair up `manL(1)` and `womanL(1)`.

The reaction we thought of,

```
go { case manL(m) + womanL(w) if m == w ⇒ beginDancing() }
```

does not actually enforce the requirement that `manL(0)` and `womanL(0)` should begin dancing first. How can we prevent the molecules `manL(1)` and `womanL(1)` from reacting if `manL(0)` and `womanL(0)` have not yet reacted?

There are two ways of achieving this in `Chymyst`:

- adding a new static molecule,
- using pipelined molecules with single-thread reactions.

Using an intermediate static molecule

We would like to prevent the dancing reaction from starting out of order. In the chemical machine paradigm, a general way to prevent reactions from starting is to omit some input molecules. Therefore, the dancing reaction needs to have *another* input molecule, say `mayBegin()`. If the dancing reaction has the form `manL + womanL + mayBegin → ...`, and if `mayBegin()` carries value 0, we can enforce the requirement that `manL(0)` and `womanL(0)` should begin dancing first.

Now it is clear that the `mayBegin` molecule must carry the most recently used position label, which will be incremented every time a new pair goes off to dance:

```
go { case manL(m) + womanL(w) + mayBegin(l) if m == w && w == l ⇒
  beginDancing(); mayBegin(l + 1)
}
```

For this to work, only a single copy of `mayBegin()` should be available in the soup. To make this intent clear, we can declare `mayBegin()` as a static molecule.

The other auxiliary molecules, `queueMen()` and `queueWomen()`, should be also emitted only once. We write a static reaction to declare all these molecules as static:

```
go { case _ ⇒ queueMen(0) + queueWomen(0) + mayBegin(0) }
```

The complete working test code is found in `Patterns01Spec.scala`.

Using pipelined molecules and single-thread pools

Pipelined molecules are stored in an ordered queue and always consumed in the FIFO order of their emission. We can implement the dancing problem with fewer molecules and reactions if we use pipelined molecules.

All emitted copies of a pipelined molecule are stored in an ordered queue, in the order they are emitted. Non-pipelined molecules are stored in an unordered multiset.

Reactions consuming a pipelined molecule are chosen by examining the molecule at the head of the queue. The result is that pipelined molecules are always consumed in the FIFO order of their emission. `Chymyst` classifies molecules automatically into pipelined and non-pipelined by checking that no deadlocks will arise if a given molecule emitter were to store the molecule instances in an ordered queue.

Consider again the reaction we started with:

```
go { case man(_) + woman(_) ⇒ beginDancing() }
```

In this reaction, both molecules `man()` and `woman()` have no guard conditions. Since this reaction is the only one consuming these molecules, they will be pipelined.

In this case, each newly emitted `man()` molecule will arrive to its queue in the order it was emitted. Another queue would hold all emitted copies of the `woman()` molecule. The reaction will automatically pick the molecules from the top of each queue.

Will this guarantee that calls to `beginDancing()` will be performed in the required order? Actually, no!

The reason is that reactions are scheduled asynchronously, on some background threads. All we have achieved so far is that the instances of the reaction will be scheduled in the correct order. However, threads may become busy or free at unpredictable times, so it is not guaranteed that reactions start in the same order they are scheduled.

To guarantee that calls to `beginDancing()` are performed in the correct order, we need to restrict this reaction to run on a *single thread* rather than (by default) in parallel, on a number of threads in the default thread pool.

This restriction does not modify the semantics of the program: it is meaningless to say that events occur in order and yet that several events may occur in parallel. If we need to restrict events to occur in a specific order, we must at the same time prohibit events from occurring in parallel.

The following code creates a single-thread pool and defines the reaction to run on that pool only:

```
val pool = FixedPool(1)

site(
  go { case man(_) + woman(_) ⇒ beginDancing() } onThreads(pool)
)
```

Test code in `Patterns01Spec.scala` verifies that this chemical program works correctly.

State machines

A state machine starts out in a certain initial state `s` of type `S` and receives actions of type `A`. The transition function `tr: (A, S) ⇒ S` defines what new state will be chosen when an action `a: A` is received in a given state `s: S`. While performing the state transition, the state machine can also execute arbitrary code for side effects.

One way of modeling a state machine is to represent the current state by a molecule `state: M[S]` and actions by molecules `action: M[A]`.

```
val a = m[A]
val s = m[S]
val tr: (A, S) ⇒ S = ???
val initialState: S = ???

site(
  go { case action(a) + state(s) ⇒ sideEffects(a, s); state(tr(a, s)) },
  go { case _ ⇒ state(initialState) }
)
```

We declared `state()` as a static molecule because, most likely, it will be necessary to guarantee that there exists only one copy of `state()` in the soup.

If the state type `S` is a disjunction type (in Scala, a disjunction type is a sealed trait extended by a fixed number of case classes), we may choose another way of modeling the state machine: Each case class in the disjunction is represented by a different molecule, and there are separate reactions for transitions involving different states. Here is an example:

```
sealed trait States
case object Initial extends States
final case class State1(x: Int) extends States
final case class State2(s: String) extends States

val stateInit = m[Initial]
val state1 = m[State1]
val state2 = m[State2]

val action = m[A]

site(
  go { case action(a) + stateInit(s) ⇒ ... state1(...) }, // whichever is appropriate
  go { case action(a) + state1(s) ⇒   ... },
  go { case action(a) + state2(s) ⇒   ... }
)

stateInit(Initial) // emit initial state
```

Here, each reaction's body is specialized to the case of the given current state. This may be a more convenient way of organizing the code.

One difference from the previous pattern is that the state-representing molecules are no longer static. Therefore, we need to guarantee that no extraneous copies of `stateInit()`, `state1()`, `state2()` can be emitted by the user code. This can be easily arranged by encapsulating the chemistry in a local function scope and by exposing only the action molecules outside that scope.

If the action type `A` is a disjunction type, we can likewise use separate molecules for representing different actions. Represented in this way, a state machine is translated into declarative code, at the cost of having to define many more molecules and reactions.

Variations on Readers/Writers

Ordered `m` : `n` Readers/Writers ("Unisex bathroom")

The Readers/Writers problem is now reformulated with a new requirement that processes should gain access to the resource in the order they requested it. The code should admit `m` concurrent Readers or `n` concurrent Writers (but not both at the same time).

We are going to model requests by emitting molecules, and we would like the molecules to be consumed in the order emitted. We can maintain the order of molecules in two ways:

- add extra molecule values and perform explicit book-keeping, as in the "dancing pairs" problem

- use pipelined molecules with single-thread reactions

Let us use the second method, hoping that the code will be more concise.

In order to handled requests via an ordered queue, we need to use a *single* pipelined molecule for all Reader and Writer requests. Clients will need to emit that molecule with appropriate values; let us call it the `request` molecule. If this molecule is pipelined, the chemical machine will consume each copy of `request` in the correct order.

The request molecule should carry a value that distinguishes Reader from Writer requests. For simplicity, we assume that the Reader and Writer requests do not pass any values but only produce some unspecified side-effect with the resource:

```
sealed trait RequestType
case object Reader extends RequestType
case object Writer extends RequestType

val request = m[RequestType]

val readerRequest = m[Unit]
val readerFinished = m[Unit]
val writerRequest = m[Unit]
val writerFinished = m[Unit]

site(
  go { case readerRequest(_) ⇒ readResource(); readerFinished() },
  go { case writerRequest(_) ⇒ writeResource(); writerFinished() }
)
```

The resource should accept Reader requests only if it is currently being accessed by no writers and by less than `m` Readers; similarly for Writer requests. Accepting a request is modeled by starting a reaction. This reaction should not start when the requirements are not met.

Since we would like the `request()` molecule to be pipelined, we cannot use any guard conditions on the reactions that consume `request()`. The only other way to control reactions is via the presence or the absence of input molecules. Therefore, we need to define another molecule, say `consume()`, that will react with the `request()` molecule. The `consume()` molecule will be present only when a new request molecule can be consumed.

```
val consume = m[Unit]
go { case request(r) + consume(_) ⇒ ??? }
```

Suppose there are already some Readers accessing the resource, and a Writer request is received. The new request cannot be processed until all Readers are done. However, the information that the new request is a Writer request is only available as a value carried by the `request()` molecule. This value remains unknown until the new `request()` molecule is consumed by some reaction.

Therefore, we need to consume the new `request()` molecule, consider this request as "pending", and then decide whether the pending request can be granted. If not, we should not consume any further `request()` molecules but instead wait for the pending request to be processed.

Therefore, we need to model the resource by a state machine whose states represent not only the following situations:

- no Readers or Writers currently working -- can accept any request; this is the initial state
- less than `m` Readers currently working, no Writer requests pending, can accept a Reader request
- some Readers currently working, a Writer request is pending, cannot accept any further requests
- exactly `m` Readers currently working, cannot accept any further requests
- less than `n` Writers currently working, no Reader requests pending, can accept a Writer request
- some Writers currently working, a Reader request is pending, cannot accept any further requests
- exactly `n` Writers currently working, cannot accept any further requests

```
val noRequests = m[Unit]
val haveReaders = m[Int]
val haveWriters = m[Int]
val haveReadersPendingWriter = m[Int]
val haveWritersPendingReader = m[Int]
val pending = m[RequestType]
```

The initial emitted molecules are `admitAny()` and `accessGiven()`. The possible state transitions are directly described by the chemical program:

```
site(
  go { case request(r) + consume(_) ⇒ pending(r) },
  go { case pending(Reader) + noRequests(_) ⇒ readerRequest() + haveReaders(1) + consu
me() },
  go { case pending(Reader) + haveReaders(k) if k < nReaders ⇒ readerRequest() + haveR
eaders(k + 1) + consume() },

  go { case pending(Writer) + noRequests(_) ⇒ writerRequest() + haveWriters(1) + consu
me() },
  go { case pending(Writer) + haveWriters(k) if k < nWriters ⇒ writerRequest() + haveW
riters(k + 1) + consume() },

  go { case pending(Writer) + haveReaders(k) ⇒ haveReadersPendingWriter(k) },
  go { case pending(Reader) + haveWriters(k) ⇒ haveWritersPendingReader(k) },

  go { case readerFinished(_) + haveReaders(k) ⇒ if (k > 1) haveReaders(k - 1) else no
Requests() },
  go { case readerFinished(_) + haveReadersPendingWriter(k) ⇒
    if (k > 1) haveReadersPendingWriter(k - 1) else {
      haveWriters(1)
      writerRequest()
      consume()
      }
    },
  go { case writerFinished(_) + haveWriters(k) ⇒ if (k > 1) haveWriters(k - 1) else no
Requests() },
  go { case writerFinished(_) + haveWritersPendingReader(k) ⇒
    if (k > 1) haveWritersPendingReader(k - 1) else {
      haveReaders(1)
      readerRequest()
      consume()
      }
    }
)
```

The complete working test is in `Patterns01Spec.scala`.

Exercise: Ordered `m` : `n` Readers/Writers that work with data

We have implemented the ordered `m` : `n` Readers/Writers problem where the read and write requests are functions without arguments returning `Unit`. Modify the code so that write requests have a `String` argument, while read requests cause an auxiliary molecule to be emitted with a `String` value.

Exercise: Ordered `m` : `n` Readers/Writers that finish their work

Add appropriate new reactions molecules such that the code can wait until a certain (fixed) number of Readers and Writers are finished with their requests.

Majority rule `n` : `n` Readers/Writers ("The Modus Hall problem")

For this example, the Readers/Writers access numbers are equal, `n` : `n`. However, a new rule involving wait times is introduced: If more Readers than Writers are waiting to access the resource, no more Writers should be granted access, and vice versa. Accordingly, we no longer require that all requests be served in the exact order received.

TODO

Fair `m` : `n` Readers/Writers

The majority rule does not guarantee fairness: If, say, on the average twice as many Reader as Writer requests arrive per unit time, it can happen that there are *always* more waiting Readers than waiting Writers. In that case, the majority rule will prevent all Writers from accessing the resource, causing "starvation" for the waiting Writers.

To fix this situation, we introduce the key new requirement that both Readers and Writers should be able to work starvation-free. Even if there is a heavy stream of Readers and a single incoming Writer, that Writer should not wait indefinitely. The algorithm should guarantee a fixed upper limit on the waiting time for both Readers and Writers, regardless of the order of arriving Readers and Writers.

The parameters `m` and `n` should allow the program to optimize its throughput when the incoming stream of Readers and Writers has the average ratio `m` : `n`. However, the order in which Readers and Writers get to work is now unimportant.

TODO

Choose and reply to one of many blocking calls (Unix `select`, Actor model's `receive`)

The task is to organize the processing of several blocking calls emitted by different concurrent reactions. One receiver is available to process one of these calls at a time.

TODO

Concurrent recursive traversal ("fork/join")

A typical task of "fork/join" type is to traverse recursively a directory that contains many files and subdirectories. For each file found by the traversal, some operation needs to be performed, such as getting the file size or computing a word count over the file's text. Finally, all the gathered data needs to be aggregated in some way: for instance, by creating a histogram of file sizes over all files.

This procedure is similar to "map/reduce" in that tasks are first split into smaller sub-tasks and then the results of all sub-tasks are aggregated. The main difference is that the "fork/join" procedure will split its sub-tasks into further sub-tasks, which may be again be split into yet smaller sub-tasks. The total number of splitting levels is known only at run time, so the procedure results in a computation tree of more or less arbitrary shape. The vertices of the tree are sub-tasks that are split; the leaves of the tree are sub-tasks that can be completed without any splitting.

Tasks of "fork/join" type can be formalized by specifying these four items:

- A type `T` representing a value that fully specifies one sub-task.
- A type `R` representing a partial result value computed by any sub-task (whether or not that subtask was split into further subtasks). We assume that `R` is a monoid type with a binary operation `++` of type `(R, R) ⇒ R`, so that we can aggregate partial results into the final result of type `R`.
- A value `init` of type `T` that specifies the main task (the root of the computation tree).
- A function `fork` of type `T ⇒ Either[List[T], R]` that decides whether the task needs to be split. If so, `fork` determines the list of new values of type `T` that correspond to the new sub-tasks. Otherwise, `fork` will compute a partial result of type `R`.

We assume for simplicity that the aggregation of partial results can be performed in any order, so that we can use a commutative monoid for `R`.

Let us now implement this generic "fork/join" procedure using `Chymyst`. The type signature should look like this:

```
def doForkJoin[R, T](init: T, fork: T ⇒ Either[List[T], R]): R = ???
```

How would we design the chemistry that performs this procedure?

It is clear that each sub-task needs to run a reaction starting with a molecule that carries a value of type `T`.

```
val task = m[T]

site( go { case task(t) ⇒ ??? } )

// Initially, emit one `task` molecule.
task(init)
```

When the reaction `task → ...` is finished, it should either emit a number of other `task` molecules, or return a result value. Therefore, we need another type of molecule that carries the result value:

```
val res = m[R]
val task = m[T]

site(
 go { case task(t) ⇒
  fork(t) match {
       case Left(ts) ⇒ ts.foreach(x ⇒ task(x))
       case Right(r) ⇒ res(r)
  }
 }
)

// Initially, emit one `task` molecule.
task(init)
```

After all `task()` molecules are consumed, a number of `res(...)` molecules will be emitted into the soup. To aggregate their values into a single final result, we need to add a reaction for `res(...)` :

```
go { case res(x) + res(y) ⇒ res(x ++ y) }
```

The result of this chemistry will be that eventually a single `res(r)` molecule will be present in the soup, and its value `r` will be the final result value. There remains, however, a major problem with the code as written so far: it does not terminate! The chemical machine does not know how many `task` molecules to expect, and so it keeps waiting for more `task(...)` molecules to be emitted into the soup.

Not knowing *when* molecules are emitted is a fundamental feature of programming in the chemical machine paradigm. That feature makes the programs robust with respect to accidental slowness of the computer, making race conditions impossible. The price is the need for additional bookkeeping in programs that need to wait until a number of tasks are finished.

In previous chapters, we have already seen two examples of this kind of bookkeeping. In the "map/reduce" pattern, we had put an additional counter on the `result` molecules so that we would know when we finish aggregating the partial results. In the "merge/sort" example, we used a chain reaction that guarantees eventual termination of the computation tree.

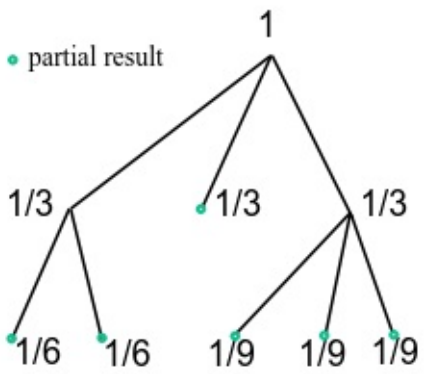
The recursive code is elegant in its way, but makes reasoning more complicated. Let us first try to achieve termination by using a counter.

If `Counter` represents the type of the counter value, we might write code similarly to the "map/reduce" pattern where we add up the counters on `res()` molecules until the total accumulated count reaches a predefined value:

```
type Counter = ???
val res = m[(R, Counter)]
val done = m[R]
val total: Counter = ???

go { case res((x, a)) + res((y, b)) ⇒
    val c = a + b
    val r = x ++ y
    if (c < total) res((r, c))
    else done(r)
}
```

However, a simple integer counter will not work in the present situation because we do not know in advance how many `task()` molecules will be generated. We need a different approach.

Consider an example where the main task is first split into `3` subtasks. We can imagine that each subtask now has to perform a fraction `1/3` of the total work. Now suppose that one of these subtasks is further split into `3`, another into `2` subtasks, and the third one is not split.



The computation tree now contains one subtask that needs to perform `1/3` of the work, 2 subtasks that need to perform `1/6` of the work, and 3 subtasks that need to perform `1/9` of the work. The sum total of the work fractions is `1/3 + 2 * 1/6 + 3 * 1/9 = 1`, as it should be.

Therefore, what we need is to assign *fractional weights* to the result values of all subtasks. The sum total of all fractional weights will remain `1`, no matter how we split tasks into subtasks. When we aggregate partial results, we will add up the fractional weights. When the weight of a partial result is equal to 1, that result is actually the total final result of the entire computation tree. In this way we can easily detect the termination of the entire task.

Thus, `Counter` must be a numerical data type that performs exact fractional arithmetic. For the present computation, we do not actually need a full implementation of fractional arithmetic; we only need to be able to add two fractions, to divide a fraction by an integer, and to compare fractions with `1`.

Assuming that this data type is available as `SimpleFraction`, we can write the "fork/join" procedure:

```
def doForkJoin[R, T](init: T, fork: T ⇒ Either[List[T], R], done: M[R]): Unit = {

  type Counter = SimpleFraction

  val res = m[(R, Counter)]
  val task = m[(T, Counter)]

  site(
    go { case task((t, c)) ⇒
      fork(t) match {
           case Left(ts) ⇒ ts.foreach(x ⇒ task((x, c / ts.length)))
           case Right(r) ⇒ res((r, c))
      }
     },
    go { case res((x, a)) + res((y, b)) ⇒
        val c = a + b
        val r = x ++ y
        if (c < 1) res((r, c))
        else done(r)
    }
  )
  // Initially, emit one `task` molecule with weight `1`.
  task((init, SimpleFraction(1)))
}
```

Exercises

Bug fixing

The code as shown in the previous section will fail in certain corner cases:

- when any task is split into an empty list of subtasks, the code will divide `c` by `ts.length`, which will be equal to zero
- when there is only one `res()` molecule ever emitted, the reaction `res + res → res` will never run; this will happen, for instance, if the initial task is split into exactly one sub-task, which then immediately returns its result

Fix the chemistry so that the procedure works correctly in these corner cases.

Ordered fork/join

In the "fork/join" chemistry just described, partial results are aggregated in an arbitrary order. Implement the chemistry using chain reactions instead of counters, so that the partial results are always aggregated first within the recursive split that generated them.

Producer-consumer, or `java.util.concurrent.ConcurrentLinkedQueue`

Unordered bag

There are many producers and consumers working with a single bag of items. Let us assume for simplicity that an item is identified by a random integer value.

Producers repeatedly add items to the bag, one item at a time. Consumers repeatedly attempt to fetch items from the bag, one item at a time. Items can be fetched from the bag in arbitrary order. If the queue is empty, the fetching operation blocks until another item is added by the producers.

TODO: expand

Ordered queue

The formulation of the problem is the same as in the unordered version, except that the bag must be replaced by a FIFO pipeline or queue: Consumers must receive fetched items in the order these items were added to the queue.

TODO: expand

Finite unordered bag

The formulation of the problem is the same as in the unordered version, except that the bag is finite and can hold no more than `n` items. If the bag already contains `n` items, a call to add another item must block until a consumer withdraws some item from the bag.

Finite ordered queue

TODO

Dining philosophers with bounded wait time

Our simple solution to the "Dining Philosophers" problem has a flaw: any given philosopher faces a theoretically unlimited waiting time in the "hungry" state. For instance, it can happen that philosopher 1 starts eating, which prevents philosopher 2 from eating. So, fork `f23` remains free and philosopher 3 could start eating concurrently if `f34` is also free.

There is a bound on the maximum time each philosopher will eat. However, by pure chance, philosophers 1 and 3 could take turns eating for any period of time (1, 3, 1, 3, etc.). While this is happening, philosopher 2 cannot start eating. The probability of waiting for any period of time is nonzero.

Let us revise the solution so that we guarantee bounded waiting time for every philosopher.

TODO

Generalized Smokers/Philosophers/Producers/Consumers problem

There are `n` sorts `A_1`, `A_2`, ..., `A_n` of items that many producers will add at random intervals and in random quantities to the common store. There are `n` sorts `P_1`, `P_2`, ..., `P_n` of consumers that look for specific sets of items. Consumers of sort `P_j` need items of sorts `a_i` for all `i` that satisfy `(j - 1 <= i <= j + 1) mod n`. Each consumer will at once fetch 3 items from the bag, if possible.

Finite resource with refill ("Dining savages")

There is a common store of items, an arbitrary number of consumers, and one refilling agent. The store can hold at most `n` items. Each consumer will try to call `fetchItem()`, but that function can be called only if the store is not empty. In that case, `fetchItem()` will

remove one item from the store.

Only if the store is empty, the `refill()` function can be called. However, only one `refill()` call can be made concurrently.

Finite server queue ("Barber shop problem")

A server can process only one job at a time by calling `processJob()`. Jobs can be submitted to the server by calling `submit()`. If the server is busy, the submitted job waits in a queue that can hold up to `n` jobs. If the queue is full, the submitted job is rejected by calling `reject()`.

Processing each job takes a finite amount of time. While the job queue is not empty, the server should keep processing jobs. When the queue is empty, the server goes into an energy-conserving "sleeping" state. When the queue becomes non-empty while the serves is sleeping, the `wakeUp()` function must be called. The server should then resume processing the jobs.

Finite ordered server queue

Same problem, but jobs must be served in the FIFO order.

Puzzles from "The Little Book of Semaphores"

The formulations of these puzzles are copied verbatim from A. Downey's book. This copying is permitted by the Creative Commons license. I will rephrase these problem formulations later, when I get to writing up their solutions.

Hilzer's "Barber shop"

Our barbershop has three chairs, three barbers, and a waiting area that can accommodate four customers on a sofa and that has standing room for additional customers. Fire codes limit the total number of customers in the shop to 20.

A customer will not enter the shop if it is filled to capacity with other customers. Once inside, the customer takes a seat on the sofa or stands if the sofa is filled. When a barber is free, the customer that has been on the sofa the longest is served and, if there are any standing customers, the one that has been in the shop the longest takes a seat on the sofa. When a

customer's haircut is finished, any barber can accept payment, but because there is only one cash register, payment is accepted for one customer at a time. The barbers divide their time among cutting hair, accepting payment, and sleeping in their chair waiting for a customer.

In other words, the following synchronization constraints apply:

Customers invoke the following functions in order: enterShop, sitOnSofa, getHairCut, pay.

Barbers invoke cutHair and acceptPayment.

Customers cannot invoke enterShop if the shop is at capacity.

If the sofa is full, an arriving customer cannot invoke sitOnSofa.

When a customer invokes getHairCut there should be a corresponding barber executing cutHair concurrently, and vice versa.

It should be possible for up to three customers to execute getHairCut concurrently, and up to three barbers to execute cutHair concurrently.

The customer has to pay before the barber can acceptPayment.

The barber must acceptPayment before the customer can exit.

The Santa Claus problem

This problem is from William Stallings's Operating Systems , but he attributes it to John Trono of St. Michael's College in Vermont.

Santa Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:

After the ninth reindeer arrives, Santa must invoke prepareSleigh, and then all nine reindeer must invoke getHitched.

After the third elf arrives, Santa must invoke helpElves. Concurrently, all three elves should invoke getHelp.

All three elves must invoke getHelp before any additional elves enter (increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

Building H2O

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

As each thread passes the barrier, it should invoke bond. You must guarantee that all the threads from one molecule invoke bond before any of the threads from the next molecule do.

In other words:

If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.

If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

We don’t have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets; thus, if we examine the sequence of threads that invoke bond and divide them into groups of three, each group should contain one oxygen and two hydrogen threads.

River crossing problem

This is from a problem set written by Anthony Joseph at U.C. Berkeley, but I don’t know if he is the original author. It is similar to the H2O problem in the sense that it is a peculiar sort of barrier that only allows threads to pass in certain combinations.

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

As each thread boards the boat it should invoke a function called board. You must guarantee that all four threads from each boatload invoke board before any of the threads from the next boatload do.

After all four threads have invoked board, exactly one of them should call a function named rowBoat, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

The roller coaster problem

Suppose there are n passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold C passengers, where C < n. The car can go around the tracks only when it is full.

Here are some additional details:

Passengers should invoke board and unboard.

The car should invoke load, run and unload.

Passengers cannot board until the car has invoked load

The car cannot depart until C passengers have boarded.

Passengers cannot unboard until the car has invoked unload.

Puzzle: Write code for the passengers and car that enforces these constraints.

Multi-car Roller Coaster problem

This solution does not generalize to the case where there is more than one car. In order to do that, we have to satisfy some additional constraints:

Only one car can be boarding at a time.

Multiple cars can be on the track concurrently.

Since cars can’t pass each other, they have to unload in the same order they boarded.

All the threads from one carload must disembark before any of the threads from subsequent carloads.

Puzzle: modify the previous solution to handle the additional constraints. You can assume that there are m cars, and that each car has a local variable named i that contains an identifier between 0 and m − 1.

The search-insert-delete problem

Three kinds of threads share access to a singly-linked list: searchers, inserters and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions.

Puzzle: write code for searchers, inserters and deleters that enforces this kind of three-way categorical mutual exclusion.

The sushi bar problem

This problem was inspired by a problem proposed by Kenneth Reek . Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.

Puzzle: write code for customers entering and leaving the sushi bar that enforces these requirements.

The child care problem

At a child care center, state regulations require that there is always one adult present for every three children.

Puzzle: Write code for child threads and adult threads that enforces this constraint in a critical section.

Optimize the child care center utilization: children can enter at any time, provided that there are enough adults present. Adults can leave at any time, provided that there are not too many children present.

The room party problem

The following synchronization constraints apply to students and the Dean of Students:

Any number of students can be in a room at the same time.

The Dean of Students can only enter a room if there are no students in the room (to conduct a search) or if there are more than 50 students in the room (to break up the party).

While the Dean of Students is in the room, no additional students may enter, but students may leave.

The Dean of Students may not leave the room until all students have left.

There is only one Dean of Students, so you do not have to enforce exclusion among multiple deans.

Puzzle: write synchronization code for students and for the Dean of Students that enforces all of these constraints.

The Senate Bus problem

Riders come to a bus stop and wait for a bus. When the bus arrives, all the waiting riders invoke boardBus, but anyone who arrives while the bus is boarding has to wait for the next bus. The capacity of the bus is 50 people; if there are more than 50 people waiting, some will have to wait for the next bus.

When all the waiting riders have boarded, the bus can invoke depart. If the bus arrives when there are no riders, it should depart immediately.

Puzzle: Write synchronization code that enforces all of these constraints.

The Faneuil Hall problem

“There are three kinds of threads: immigrants, spectators, and a one judge. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization. After the

confirmation, the immigrants pick up their certificates of U.S. Citizenship. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave."

To make these requirements more specific, let's give the threads some functions to execute, and put constraints on those functions.

Immigrants must invoke enter, checkIn, sitDown, swear, getCertificate and leave.

The judge invokes enter, confirm and leave.

Spectators invoke enter, spectate and leave.

While the judge is in the building, no one may enter and immigrants may not leave.

The judge can not confirm until all immigrants who have invoked enter have also invoked checkIn.

Immigrants can not getCertificate until the judge has executed confirm.

Solve this.

Extended version: modify this solution to handle the additional constraint that after the judge leaves, all immigrants who have been sworn in must leave before the judge can enter again.

Dining Hall problem

Students in the dining hall invoke dine and then leave. After invoking dine and before invoking leave a student is considered "ready to leave".

The synchronization constraint that applies to students is that, in order to maintain the illusion of social suave, a student may never sit at a table alone. A student is considered to be sitting alone if everyone else who has invoked dine invokes leave before she has finished dine.

Puzzle: write code that enforces this constraint.

Extended Dining Hall problem

The Dining Hall problem gets a little more challenging if we add another step. As students come to lunch they invoke getFood, dine and then leave. After invoking getFood and before invoking dine, a student is considered "ready to eat". Similarly, after invoking dine a student is considered "ready to leave".

The same synchronization constraint applies: a student may never sit at a table alone. A student is considered to be sitting alone if either

She invokes dine while there is no one else at the table and no one ready to eat, or

everyone else who has invoked dine invokes leave before she has finished dine.

Puzzle: write code that enforces these constraints.

Cigarette smokers

The "Cigarette smokers" problem is to implement four concurrent processes that coordinate assembly line operations to manufacture cigarettes with the workers smoking the individual cigarettes they manufacture. One process represents a supplier of ingredients on the assembly line, which we may call the pusher. The other processes are three smokers, each having an infinite supply of only one of the three required ingredients, which are matches, paper, and tobacco. We may give names to the smokers/workers as Keith, Slash, and Jimi.

The pusher provides at random time intervals one unit each of two required ingredients (for example matches and paper, or paper and tobacco). The pusher is not allowed to coordinate with the smokers to use knowledge of which smoker needs which ingredients, he just supplies two ingredients at a time. We assume that the pusher has an infinite supply of the three ingredients available to him. The real life example is that of a core operating systems service having to schedule and provide limited distinct resources to other services where coordination of scarce resources is required.

Each smoker selects two ingredients, rolls up a cigarette using the third ingredient that complements the list, lights it up and smokes it. It is necessary for the smoker to finish his cigarette before the pusher supplies the next two ingredients (a timer can simulate the smoking activity). We can think of the smoker shutting down the assembly line operation until he is done smoking.

We model the processes as follows:

Supplier		Smoker 1		Smoker 2		Smoker 3
select 2 random ingredients		pick tobacco and paper		pick tobacco and matches		pick matches and paper

Let us now figure out the chemistry that will solve this problem. We can think of the problem as a concurrent producer consumer queue with three competing consumers and the supplier simply produces random pairs of ingredients into the queue. For simplicity, we assume the queue has capacity 1, which is an assumption in the statement of the problem (smoker shuts down factory operation while he takes a smoke break, thus pausing the pusher).

It is important to think of a suitable data model to capture the state of the world for the problem, so we need to know when to stop and count how many cycles we go through, if we want to stop the computation. It may be useful to keep track of how many ingredients have been shipped or consumed but this does not look to be important for a minimal solution. We include inventory tracking because of some logging we do to represent the concurrent activities more explicitly; this inventory tracking does add a bit of complexity.

For counting, we will use a distinct molecule `count` dedicated just to that and emit it initially with a constant number. We also use a blocking molecule `check` that we emit when we reach a `count` of 0. This approach is same as in several other examples discussed here.

```
val count = m[Int]
val check = b[Unit, Unit]

site(tp) ( // reactions
  go { case pusher(???) + count(n) if n >= 1 => ??? // the supply reaction TBD
       count(n-1) // let us include this decrement now.
  },
  go { case count(0) + check(_, r) => r() }  // note that we use mutually exclusive co
nditions on count in the two reactions.
)
// emission of initial molecules in chemistry follows
// other molecules to emit as necessary for the specifics of this problem
count(supplyLineSize) // if running as a daemon, we would not use the count molecule a
nd let the example/application run for ever.
check()
```

We now introduce one molecule for each smoker and their role should be symmetrical while capturing the information about their ingredient input requirements. Let us give names to smoker molecules as `Keith`, `Slash`, and `Jimi`. Let us assign one molecule per ingredient: `paper`, `matches`, and `tobacco`. This will represent the last three reactions. We need to emit the molecules for `Keith`, `Slash`, and `Jimi` on start up, which can be combined (`Keith(()) + Slash(()) + Jimi(())`).

Here, we write up a helper function `enjoyAndResume` that is a refactored piece of code that is common to the three smokers, who, when receiving ingredients, make the cigarette, smoke it while taking a break off work, shutting down the operations and then resuming operation to `pusher` when done, which is required as the smoker must notify the `pusher` when to resume. The smoking break is a simple waste of time represented by a sleep. The smoker molecule must re-emit itself once done to indicate readiness of the smoker to get back to work to process the next pair of ingredients.

Notice here that we capture a state of the shipped inventory from the ingredient molecules, all of which carry the value of the inventory and echo it back to the pusher so that he knows where he is at in his bookkeeping; the smokers collaborate and don't lie so simply echo back the inventory as is (note that this is not necessary if we are not interested in tracking down how many ingredients have been used in manufacturing).

```
def smokingBreak(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble*20.0 +
2.0).toLong)
def enjoyAndResume(s: ShippedInventory) = {
   smokingBreak()
   pusher(s)
}
site(tp) ( // reactions
  // other reactions ...
   go { case Keith(_) + tobacco(s) + matches(_) => enjoyAndResume(s); Keith() },
   go { case Slash(_) + tobacco(s) + paper(_) => enjoyAndResume(s); Slash() },
   go { case Jimi(_) + matches(s) + paper(_) => enjoyAndResume(s); Jimi()}
)
// other initial molecules to be emitted
Keith() + Slash() + Jimi()
```

Now, we need the `pusher` molecule to generate a pair of ingredients randomly at time intervals. Paying attention to the statement of the problem, we notice that he needs to wait for a smoker to be done, hence as stated before, we simply need to emit the `pusher` molecule from the smoker reactions (a signal in conventional terminology) and the `pusher` molecule should be emitted on start up and should respond to the count molecule to evaluate the deltas in the `ShippedInventory`; the active `pusher` can be thought of as representing an active factory so we must emit its molecule on start up.

We represent the shipped inventory as a case class with a count for each ingredient, we call it `ShippedInventory`. We integrate counting as previously discussed, introduce the random selection of ingredient pairs and emit the molecules for the pair of ingredients.

```
case class ShippedInventory(tobacco: Int, paper: Int, matches: Int)
site(tp) (
    go { case pusher(ShippedInventory(t, p, m)) + count(n) if n >= 1 =>
      scala.util.Random.nextInt(3) match { // select the 2 ingredients randomly
        case 0 =>
          val s = ShippedInventory(t+1, p, m+1)
          tobaccoShipment(s)
          matchesShipment(s)
        case 1 =>
          val s =  ShippedInventory(t+1, p+1, m)
          tobaccoShipment(s)
          paperShipment(s)
        case _ =>
          val s = ShippedInventory(t, p+1, m+1)
          matchesShipment(s)
          paperShipment(s)
      }
      count(n-1)
    }
)
count(supplyLineSize)
pusher(ShippedInventory(0,0,0))
// other initial molecules to be emitted (the smokers and check)
```

The final code looks like this:

```
val supplyLineSize = 10
def smokingBreak(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble*20.0 +
2.0).toLong)

case class ShippedInventory(tobacco: Int, paper: Int, matches: Int)
// this data is only to demonstrate effects of randomization on the supply chain and m
ake content of logFile more interesting.
// strictly speaking all we need to keep track of is inventory. Example would work if
pusher molecule value would carry Unit values instead.

val pusher = m[ShippedInventory] // pusher means drug dealer, in classic Comp Sci, we'
d call this producer or publisher.
val count = m[Int]
val KeithInNeed = new M[Unit]("Keith obtained tobacco and matches to get his fix") //
makes for more vivid tracing, could be plainly m[Unit]
val SlashInNeed = new M[Unit]("Slash obtained tobacco and matches to get his fix") //
same
val JimiInNeed = new M[Unit]("Jimi obtained tobacco and matches to get his fix") // sa
me

val tobaccoShipment = m[ShippedInventory] // this is not particularly elegant, ideally
 this should carry Unit but pusher needs to obtain current state
val matchesShipment = m[ShippedInventory] // same
val paperShipment = m[ShippedInventory] // same
```

```
val check = b[Unit, Unit] // blocking Unit, only blocking molecule of the example.

val logFile = new ConcurrentLinkedQueue[String]

site(tp) (
  go { case pusher(ShippedInventory(t, p, m)) + count(n) if n >= 1 =>
    logFile.add(s"$n,$t,$p,$m") // logging the state makes it easier to see what's goi
ng on, curious user may put println here instead.
    scala.util.Random.nextInt(3) match { // select the 2 ingredients randomly
      case 0 =>
        val s = ShippedInventory(t+1, p, m+1)
        tobaccoShipment(s)
        matchesShipment(s)
      case 1 =>
        val s =  ShippedInventory(t+1, p+1, m)
        tobaccoShipment(s)
        paperShipment(s)
      case _ =>
        val s = ShippedInventory(t, p+1, m+1)
        matchesShipment(s)
        paperShipment(s)
    }
    count(n-1)
  },
  go { case count(0) + check(_, r) => r() },

  go { case KeithInNeed(_) + tobaccoShipment(s) + matchesShipment(_) =>
    smokingBreak(); pusher(s); KeithInNeed()
  },
  go { case SlashInNeed(_) + tobaccoShipment(s) + paperShipment(_) =>
    smokingBreak(); pusher(s); SlashInNeed()
  },
  go { case JimiInNeed(_) + matchesShipment(s) + paperShipment(_) =>
    smokingBreak(); pusher(s); JimiInNeed()
  }
)

KeithInNeed() + SlashInNeed() + JimiInNeed()
pusher(ShippedInventory(0,0,0))
count(supplyLineSize) // if running as a daemon, we would not use count and let the ex
ample/application run for ever.

check()
```

There is a harder, more general treatment of the cigarette smokers problem, which has the `pusher` in charge of the rate of availability of ingredients, not having to wait for smokers, which the reader may think of using a producer-consumer queue with unlimited buffer in classic treatment of the problem. The solution is provided in code. Let us say that the

change is very simple, we just need to have pusher emit the pusher molecule instead of the smokers doing so. The pausing in the assembly line needs to be done within the `pusher` reaction, otherwise it is the same solution.

Readers Writer Locks

The problem here is to give access to a shared resource where only one thread can write to a resource at any time and multiple threads can read data from the same resource concurrently, provided that the writer thread and the reader threads are not in contention for the same resource. In other words, the writer thread can get access to the resource only if no reader thread has access to the resource and a reader thread can get access only if the writer thread has no access. This concurrency scenario has applicability in memory/disk pages in operating systems or databases; standard terminology use exclusive write lock and shared read locks.

The main idea is to use a single writer molecule and a collection of reader molecules, each of which is distinguished by a distinct name or key. Accordingly, we need molecules `val reader = m[String]` and `val writer = m[String]` ; we then need to emit these molecules as follows `readers.foreach(n => reader(n))` with `readers` being an arbitrary collection of distinct names and `writer("exclusive-writer")` . So we emit multiple `reader` molecules and a single `writer` molecule into the soup. The reactions will have to re-emit any of these molecules each time the reaction consumes one.

We need to log events as we go along. What needs to be captured is the identity of the molecule, its name, and the action taken by the molecule acquiring a lock or releasing a lock. *We ensure that there is no name collision among the `reader` molecules and the `writer` molecule.* The `LockEvent` will have a `toString` method to enable debugging or troubleshooting. We also need to block a current thread and simulate access to a critical section of code, which we do with methods `visitCriticalSection` and `leaveCriticalSection` tracking such events in a `logFile = new ConcurrentLinkedQueue[LockEvent]` for debugging or unit testing.

```
sealed trait LockEvent {
  val name: String
  def toString: String
}
case class LockAcquisition(override val name: String) extends LockEvent {
  override def toString: String = s"$name enters critical section"
}
case class LockRelease(override val name: String) extends LockEvent {
  override def toString: String = s"$name leaves critical section"
}
val logFile = new ConcurrentLinkedQueue[LockEvent]

def useResource(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble * 4.0 +
1.0).toLong)
def visitCriticalSection(name: String): Unit = {
  logFile.add(LockAcquisition(name))
  useResource()
}
def leaveCriticalSection(name: String): Unit = {
  logFile.add(LockRelease(name))
  ()
}
```

We need to consider the ending of the simulation, which we do with a `count = m[Int]` non-blocking molecule and a `check = b[Unit, Unit]` molecule as is often done here. We make the arbitrary decision that we will count down number of lock acquisitions by the `writer` molecule in way similar to the supplier or pusher for the cigarettes problem.

Now, the fun begins: we need to model how many readers are using the resource concurrently, which we do with molecule `readerCount[Int]`. So let us start with a draft, using some count down logic with a `writer` molecule reaction, ignoring helper functions and case classes we introduced already:

```
val count = m[Int]
val readerCount = m[Int]
val check = b[Unit, Unit]
val readers = "ABCDEFGH".toCharArray.map(_.toString).toVector // vector of letters as
Strings.

val reader = m[String]
val writer = m[String]

site(tp)(
  go { case writer(name) + readerCount(0) + count(n) if n > 0 =>
    visitCriticalSection(name)
    writer(name)
    count(n - 1)
    readerCount(0)
    leaveCriticalSection(name)
  },
  go { case count(0) + readerCount(0) + check(_, r) => r() }, // readerCount(0) condit
ion ensures we end when all locks are released.

  go { case readerCount(n) + reader(name)  =>
    readerCount(n+1)
    visitCriticalSection(name)
    readerCount(n - 1)
    leaveCriticalSection(name) // undefined count
    reader(name)
  }
)
readerCount(0)
readers.foreach(n => reader(n))
val writerName = "exclusive-writer"
writer(writerName)
count(supplyLineSize)

check()
```

It does not even compile! Macros code tell us *Unconditional livelock: Input molecules should not be a subset of output molecules, with all trivial matchers for* `(readerCount, reader) go { case readerCount(n) + reader(name)` . What went wrong? Well, yes, sure enough, we consume two molecules and emit three! `readerCount(n+1)` and `readerCount(n - 1)` with `reader(name)` count as three. Let us introduce an intermediate molecule in between the emission of the two `readerCount` emissions as `readerExit = m[String]` , which we do not emit into soup on start up:

```
val count = m[Int]
val readerCount = m[Int]

val check = b[Unit, Unit]

val readers = "ABCDEFGH".toCharArray.map(_.toString).toVector // vector of letters as
Strings.

val readerExit = m[String]
val reader = m[String]
val writer = m[String]

site(tp)(
  go { case writer(name) + readerCount(0) + count(n) if n > 0 =>
    visitCriticalSection(name)
    writer(name)
    count(n - 1)
    readerCount(0)
    leaveCriticalSection(name)
  },
  go { case count(0) + readerCount(0) + check(_, r) => r() }, // readerCount(0) condit
ion ensures we end when all locks are released.

  go { case readerCount(n) + readerExit(name)  =>
    readerCount(n - 1)
    leaveCriticalSection(name) // undefined count
    reader(name)
  },
  go { case readerCount(n) + reader(name)  =>
    readerCount(n+1)
    visitCriticalSection(name)
    readerExit(name)
  }
)
readerCount(0)
readers.foreach(n => reader(n))
val writerName = "exclusive-writer"
writer(writerName)
count(supplyLineSize)

check()
```

The simulation does not stop... We could replace `visitCriticalSection` and `leaveCriticalSection` with some tracing. What we see is that the `reader` molecules are reacting all the time continuously but the `writer` molecule never does. This is a starvation problem, the site is always consuming the `reader` molecule reactions as it gets data all the time and the readerCount might not reach a value of 0.

Let us have an exiting reader molecule yield by waiting for more incoming work to arrive before getting itself to read again, so we introduce `waitForUserRequest()` before emitting `reader(name)`:

```
go { case readerCount(n) + readerExit(name)  =>
  readerCount(n - 1)
  leaveCriticalSection(name) // undefined count
  waitForUserRequest() // gives a chance to writer to do some work
  reader(name)
}
```

It now works, so let's assemble the complete solution ignoring unit testing, which can be found in code. Unit test verifies no reader lock acquisition while a writer lock is active and no double locking by any lock prior to releasing.

```
val supplyLineSize = 25 // make it high enough to try to provoke race conditions, but
not so high that sleeps make the test run too slow.

sealed trait LockEvent {
  val name: String
  def toString: String
}
case class LockAcquisition(override val name: String) extends LockEvent {
  override def toString: String = s"$name enters critical section"
}
case class LockRelease(override val name: String) extends LockEvent {
  override def toString: String = s"$name leaves critical section"
}
val logFile = new ConcurrentLinkedQueue[LockEvent]

def useResource(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble * 4.0 +
1.0).toLong)
def waitForUserRequest(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble
* 4.0 + 1.0).toLong)
def visitCriticalSection(name: String): Unit = {
  logFile.add(LockAcquisition(name))
  useResource()
}
def leaveCriticalSection(name: String): Unit = {
  logFile.add(LockRelease(name))
  ()
}

val count = m[Int]
val readerCount = m[Int]

val check = b[Unit, Unit] // blocking Unit, only blocking molecule of the example.

val readers = "ABCDEFGH".toCharArray.map(_.toString).toVector // vector of letters as
Strings.
```

```
// Making readers a large collection introduces lots of sleeps since we count number o
f writer locks for simulation and the more readers we have
// the more total locks and total sleeps simulation will have.

val readerExit = m[String]
val reader = m[String]
val writer = m[String]

site(tp)(
  go { case writer(name) + readerCount(0) + count(n) if n > 0 =>
    visitCriticalSection(name)
    writer(name)
    count(n - 1)
    readerCount(0)
    leaveCriticalSection(name)
  },
  go { case count(0) + readerCount(0) + check(_, r) => r() }, // readerCount(0) condit
ion ensures we end when all locks are released.

  go { case readerCount(n) + readerExit(name)  =>
    readerCount(n - 1)
    leaveCriticalSection(name)
    waitForUserRequest() // gives a chance to writer to do some work
    reader(name)
  },
  go { case readerCount(n) + reader(name)  =>
    readerCount(n+1)
    visitCriticalSection(name)
    readerExit(name)
  }
)
readerCount(0)
readers.foreach(n => reader(n))
val writerName = "exclusive-writer"
writer(writerName)
count(supplyLineSize)

check()
```

Dining savages

We have a tribe of savages that practises cannibalism. Accordingly, the tribe is assumed to have a large supply of prisoners or victims that will make it into a pot for communal eating. The concurrency rules are as follows. The pot has capacity for `m` victims at the most. It is generally assumed that there is an arbitrary number `k` of savages who may take turn

eating one serving from the pot, which equates to a single victim. In the current presentation, we will assume that we have three eating savages, so we may associate a particular chemical reaction to a single eating savage.

The savages may start taking turn eating only when the cook is not busy, which is from the time the pot is at full capacity until it is empty. Once the pot is empty, the savages wait for the pot to be replenished, which requires the cook to add one victim at a time into the pot.

As usual, we limit the simulation for some amount of time, here we choose a parameter `n` for the number of victims being consumed by the savages. For simplicity, here, we assume that `n` is a multiple of the pot capacity `m`.

The approach to solving the problem is to start with few molecules and model the problem more precisely starting with simplifying assumptions; in particular, it is easier to think of a single savage eating from the pot and generalize to `k` afterwards.

Hint: use a counter for filling the pot and one for consuming from it

In this presentation, we will log distinct events representative of the story into a concurrent queue of events sharing a printable (String) interface. This makes it easier to visualize a number of runs, debug, and provide some assertions at the end of the simulation. If the event is unique and parameter free, we use a case object, otherwise a case class. We will show this logging aspect only at the end of the solution as it does not provide any particular insight to concurrency modeling with Chymyst.

We can start with the parameters of the problem and a simulation error condition with a pot starting full and savages never eating, instead the amount of victims in the pot decays with time randomly, warming ourselves up to solving the specifics of the problem:

```
val maxPerPot = 7
// enemies of the tribe put together in a pot or capacity of pot in number of ingredie
nts
val batches = 10
val supplyLineSize = maxPerPot * batches
val check = b[Unit, Unit]
val endSimulation = m[Unit]
val availableIngredientsInPot = m[Int]

sealed trait StoryEvent
val userStory = new ConcurrentLinkedQueue[StoryEvent]

def eatSingleServing(batchVictim: Int): Unit = {
  // userStory.add(VictimIsConsumedFromPot(batchVictim)) the parameter here is just an
 identifier
  // from the victims/ingredients in the pot starting with a high number
  Thread.sleep(math.floor(scala.util.Random.nextDouble * 20.0 + 2.0).toLong)
  availableIngredientsInPot(batchVictim - 1) // one fewer serving.
}

site(tp)(
  go { case endSimulation(_) + check(_, r) => r() },
  go { case availableIngredientsInPot(n) =>
    // userStory.add(EndOfSimulation) (adding to concurrentQueue userStory
    if (n > 0) {
      eatSingleServing(n)
    } else endSimulation()
  }
)
availableIngredientsInPot(maxPerPot) // this molecule signifies pot is available for s
avages to eat.
check()
```

Now, we need to look at a molecule that will understand to run enough cycles of pot filling up and emptying itself. A `Cook` molecule carrying out an integer number from `supplyLineSize = maxPerPot * batches` should do. It will have to count down. The `check` molecule represent the end of simulation, which will occur once the cook has enough and the pot is empty, meaning the cook has gone through enough batches to have enough. When the cook's counter is at 0, he has had enough (second reaction below), however he will leave the pot full as he found it (the initial condition for `availableIngredientsInPot`) . In the third reaction, we start filling the pot with the molecule showing cook is busy adding to the pot for a particular batch `busyCookingIngredientsInPot` .

```
go { case CookHadEnough(_) + availableIngredientsInPot(0) + check(_, r) => r()},
go { case Cook(0) =>
      CookHadEnough()
      availableIngredientsInPot(maxPerPot)
},
go { case Cook(n) + availableIngredientsInPot(0) if n > 0 => // cook gets activated on
ce the pot reaches an empty state.
  pauseForIngredient()
  busyCookingIngredientsInPot(1) // switch of counting from availableIngredientsInPot
to busyCooking indicates we're refilling the pot.
  Cook(n - 1)
}
```

We need however to have the cook fill the pot completely and so we need a new reaction to increment the ingredients to the cooking batch size, so a counting reaction for `busyCookingIngredientsInPot`. While counting up the ingredients in the batch, we must continue to count down the number of victims in the simulation. Once the cook finishes his batch and can no longer add any, the cook signifies to savages that they can eat by emitting `availableIngredientsInPot(maxPerPot)`.

We can now write

```
go { case Cook(m) + busyCookingIngredientsInPot(n) if m > 0 =>
  if (n < maxPerPot) {
    pauseForIngredient()
    busyCookingIngredientsInPot(n + 1)
    Cook(m - 1)
  } else {
    availableIngredientsInPot(maxPerPot) // switch of counting from busyCooking to ava
ilableIngredientsInPot indicates we're consuming the pot.
    Cook(m)
  }
}
```

Running through this, we run into a problem with

```
go { case availableIngredientsInPot(n) =>
  if (n > 0) {
    eatSingleServing(n)
  } else endSimulation()
}
```

and need to introduce a `savage` molecule to consume the ingredients

```
go { case savage(_) + availableIngredientsInPot(n) =>
  if (n > 0) {
    eatSingleServing(n)
    savage()
  }
}
```

We are now ready to augment the solution with a single savage:

```
val maxPerPot = 7
// enemies of the tribe put together in a pot or capacity of pot in number of ingredie
nts
val batches = 10
val supplyLineSize = maxPerPot * batches
val check = b[Unit, Unit]
val endSimulation = m[Unit]
val availableIngredientsInPot = m[Int]

val Cook = m[Int] // counts ingredients consumed, so after a while decides it's enough.

val CookHadEnough = m[Unit]
val busyCookingIngredientsInPot = m[Int] // Cook's counter to add ingredients to the p
ot
val savage = m[Unit]

sealed trait StoryEvent
val userStory = new ConcurrentLinkedQueue[StoryEvent]

def eatSingleServing(batchVictim: Int): Unit = {
  // userStory.add(VictimIsConsumedFromPot(batchVictim)) the parameter here is just an
 identifier
  // from the victims/ingredients in the pot starting with a high number
  Thread.sleep(math.floor(scala.util.Random.nextDouble * 20.0 + 2.0).toLong)
  availableIngredientsInPot(batchVictim - 1) // one fewer serving.
}

def pauseForIngredient(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble
* 20.0 + 2.0).toLong)

site(tp)(
  go { case Cook(0) =>
    CookHadEnough()
    availableIngredientsInPot(maxPerPot)
  },
  go { case CookHadEnough(_) + availableIngredientsInPot(0) + check(_, r) => r()},
  go { case savage(_) + availableIngredientsInPot(n) =>
    if (n > 0) {
      eatSingleServing(n)
      savage()
    }
  },
```

```
  go { case Cook(n) + availableIngredientsInPot(0) if n > 0 => // cook gets activated
once the pot reaches an empty state.
    pauseForIngredient()
    busyCookingIngredientsInPot(1) // switch of counting from availableIngredientsInPo
t to busyCooking indicates we're refilling the pot.
    Cook(n - 1)
  },

  go { case Cook(m) + busyCookingIngredientsInPot(n) if m > 0 =>
    if (n < maxPerPot) {
      pauseForIngredient()
      busyCookingIngredientsInPot(n + 1)
      Cook(m - 1)
    } else {
      availableIngredientsInPot(maxPerPot) // switch of counting from busyCooking to a
vailableIngredientsInPot indicates we're consuming the pot.
      Cook(m)
    }
  }

)
Cook(supplyLineSize)
availableIngredientsInPot(maxPerPot) // this molecule signifies pot is available for s
avages to eat.
savage()
check()
```

At this point, we can replace the reaction with `savage` with one reaction per savage participating in the meal, with molecules `Ivan`, `Patrick` and `Anita`. We can also introduce logging to capture events.

```
val maxPerPot = 7 // enemies of the tribe put together in a pot or capacity of pot in
number of ingredients
val batches = 10
val supplyLineSize = maxPerPot * batches
val check = b[Unit, Unit]

sealed trait StoryEvent {
  def toString: String
}
case object CookRetires extends StoryEvent {
  override def toString: String = "cook is done, savages may eat last batch"
}
case object CookStartsToWork extends StoryEvent {
  override def toString: String = "cook finds empty pot and gets to work"
}
case object EndOfSimulation extends StoryEvent {
  override def toString: String =
    "ending simulation, no more ingredients available, savages will have to fish or ea
t berries or raid again"
```

```
}
final case class CookAddsVictim(victimsToBeCooked: Int, batchVictim: Int) extends Stor
yEvent {
  override def toString: String =
    s"""cook finds unfilled pot and gets cracking with $batchVictim-th enemy ingredien
t"
    | "for current batch with $victimsToBeCooked victims to be cooked""""
}
final case class CookCompletedBatch(victimsToBeCooked: Int) extends StoryEvent {
  override def toString: String =
    s"cook notices he finished adding all ingredients with $victimsToBeCooked victims
to be cooked"
}
final case class SavageEating(name: String, batchVictim: Int) extends StoryEvent {
  override def toString: String = s"$name about to eat ingredient # $batchVictim"
}

val Cook = m[Int] // counts ingredients consumed, so after a while decides it's enough.

val CookHadEnough = m[Unit]
val busyCookingIngredientsInPot = m[Int]
val Ivan = m[Unit]   // a savage (consumer)
val Patrick = m[Unit] // a savage
val Anita = m[Unit] // a savage
val availableIngredientsInPot = m[Int]

val userStory = new ConcurrentLinkedQueue[StoryEvent]

def pauseForIngredient(): Unit = Thread.sleep(math.floor(scala.util.Random.nextDouble
* 20.0 + 2.0).toLong)
def eatSingleServing(savage: String, batchVictim: Int): Unit = {
  userStory.add(SavageEating(savage, batchVictim))
  Thread.sleep(math.floor(scala.util.Random.nextDouble * 20.0 + 2.0).toLong)
  availableIngredientsInPot(batchVictim - 1) // one fewer serving.
}

site(tp)(
  go { case Cook(0) =>
    userStory.add(CookCompletedBatch(0))
    userStory.add(CookAddsVictim(0, 1))
    userStory.add(CookRetires)
    CookHadEnough()
    availableIngredientsInPot(maxPerPot)
  },
  go { case CookHadEnough(_) + availableIngredientsInPot(0) + check(_, r) =>
    userStory.add(EndOfSimulation)
    r()
  },
  go { case Cook(n) + availableIngredientsInPot(0) if n > 0 => // cook gets activated
once the pot reaches an empty state.
    userStory.add(CookStartsToWork)
    pauseForIngredient()
    busyCookingIngredientsInPot(1) // switch of counting from availableIngredientsInPo
```

```
t to busyCooking indicates we're refilling the pot.
    Cook(n - 1)
  },

  go { case Cook(m) + busyCookingIngredientsInPot(n) if m > 0 =>
    userStory.add(CookAddsVictim(m, n))
    if (n < maxPerPot) {
      pauseForIngredient()
      busyCookingIngredientsInPot(n + 1)
      Cook(m - 1)
    } else {
      userStory.add(CookCompletedBatch(m))
      availableIngredientsInPot(maxPerPot) // switch of counting from busyCooking to a
vailableIngredientsInPot indicates we're consuming the pot.
      Cook(m)
    }
  },
  go { case Ivan(_) + availableIngredientsInPot(n) if n > 0 => eatSingleServing("Ivan"
, n) + Ivan()  },
  go { case Patrick(_) + availableIngredientsInPot(n) if n > 0 => eatSingleServing( "P
atrick", n) + Patrick()  },
  go { case Anita(_) + availableIngredientsInPot(n) if n > 0 => eatSingleServing( "Ani
ta", n) + Anita()  }

)
Patrick() + Ivan() + Anita() + Cook(supplyLineSize) // if running as a daemon, we woul
d not count down for the Cook.
availableIngredientsInPot(maxPerPot) // this molecule signifies pot is available for s
avages to eat.
check()
```

To generalize the chemistry to an arbitrary population of savages we introduce an indexed sequence of `savages` containing names or Strings. We then replace the three molecules for Ivan, Patrick, and Anita of type `m[Unit]` with a single molecule `savage` carrying a name value (`m[String]`).

Instead of emitting the three savage molecules at initialization, we emit all savages molecules as `savages.foreach(savage)` . Finally, we generalize the three specific savage reactions into the following one, selecting a random savage to emit (take turn) once the current savage is done eating:

```
go { case savage(name) + availableIngredientsInPot(n) if n > 0 =>
     eatSingleServing(name, n)
     val randomSavageName = savages(scala.util.Random.nextInt(savages.size))
     savage(randomSavageName) // emit random savage molecule
   }
```

Full code is available in examples.

Game of Life

Let us implement Conway's famous Game of Life as a concurrent computation in the chemical machine.

Our goal is to make use of concurrency as much as possible. An elementary computation in the Game of Life is to determine the next state of a cell, given its present state and the present states of its 8 neighbor cells.

```
def getNewState(
  state0: Int,
  state1: Int,
  state2: Int,
  state3: Int,
  state4: Int,
  state5: Int,
  state6: Int,
  state7: Int,
  state8: Int
): Int =
  (state1 + state2 + state3 + state4 +
     state5 + state6 + state7 + state8) match {
    case 2 ⇒ state0
    case 3 ⇒ 1
    case _ ⇒ 0
  }
```

Here, all "states" are integers `0` or `1`, and `state0` represents the center of a 3 x 3 square.

We would like this computation to proceed concurrently for each cell on the board, as much as possible. So let us make this computation into a reaction. For this to work, the previous states of each cell should be values carried by some input molecules of the reaction. Therefore, we need 9 input molecules in each reaction, each carrying the state of one neighbor cell.

First solution: single reaction

To prototype the reactions, let us imagine that we are computing the next state of the board at coordinates `(x, y)` at time `t`. Suppose that molecule `c0()` carries the state of the board at the cell `(x, y)`, while the additional molecules `c1()`, ..., `c8()` carry the

"neighbor data", that is, the states of the 8 neighbor cells at coordinates `(x - 1, y - 1)`, `(x - 1, y)`, `(x, y + 1)` and so on.

Let us assign (arbitrarily) the neighbor values like this:

`c1((x, y))`	`c2((x, y))`	`c3((x, y))`
`c4((x, y))`	`c0((x, y))`	`c5((x, y))`
`c6((x, y))`	`c7((x, y))`	`c8((x, y))`

For now, let us also put the time coordinate `t` as a value onto the molecules. Given these molecules, we can start writing a reaction like this:

```
go { case
      c0((x, y, t, state0)) +
      c1((x, y, t, state1)) +
      c2((x, y, t, state2)) +
      c3((x, y, t, state3)) +
      c4((x, y, t, state4)) +
      c5((x, y, t, state5)) +
      c6((x, y, t, state6)) +
      c7((x, y, t, state7)) +
      c8((x, y, t, state8)) ⇒ ??? }
```

Here, `c1((x, y, t, s))` represents the state of the "first" neighbor at the `(x, y)`. The values `x` and `y` always represent the coordinates of the center cell.

Now, we will immediately recognize that the reaction cannot work as written: Scala does not allow repeated pattern variables in a `case` pattern.

Our intention was to start the reaction only when all 9 input molecules have the same values of `x`, `y`, `t`. To do this, we must use a guard condition on the reaction:

```
go { case
  c0((x0, y0, t0, state0)) +
  c1((x1, y1, t1, state1)) +
  c2((x2, y2, t2, state2)) +
  c3((x3, y3, t3, state3)) +
  c4((x4, y4, t4, state4)) +
  c5((x5, y5, t5, state5)) +
  c6((x6, y6, t6, state6)) +
  c7((x7, y7, t7, state7)) +
  c8((x8, y8, t8, state8))
   if x0 == x1 && x0 == x2 && x0 == x3 && x0 == x4 &&
      x0 == x5 && x0 == x6 && x0 == x7 && x0 == x8 &&
      y0 == y1 && y0 == y2 && y0 == y3 && y0 == y4 &&
      y0 == y5 && y0 == y6 && y0 == y7 && y0 == y8 &&
      t0 == t1 && t0 == t2 && t0 == t3 && t0 == t4 &&
      t0 == t5 && t0 == t6 && t0 == t7 && t0 == t8 ⇒ ???
}
```

What should be the output molecules of that reaction? We need to compute the new state at `(x, y)` and put it onto the output molecule `c0((x, y, t + 1))`:

```
go { case
  c0((x0, y0, t0, state0)) +
  c1((x1, y1, t1, state1)) +
  c2((x2, y2, t2, state2)) +
  c3((x3, y3, t3, state3)) +
  c4((x4, y4, t4, state4)) +
  c5((x5, y5, t5, state5)) +
  c6((x6, y6, t6, state6)) +
  c7((x7, y7, t7, state7)) +
  c8((x8, y8, t8, state8))
   if x0 == x1 && x0 == x2 && x0 == x3 && x0 == x4 &&
      x0 == x5 && x0 == x6 && x0 == x7 && x0 == x8 &&
      y0 == y1 && y0 == y2 && y0 == y3 && y0 == y4 &&
      y0 == y5 && y0 == y6 && y0 == y7 && y0 == y8 &&
      t0 == t1 && t0 == t2 && t0 == t3 && t0 == t4 &&
      t0 == t5 && t0 == t6 && t0 == t7 && t0 == t8 ⇒
  val newState = getNewState(state0, state1, state2, state3,
    state4, state5, state6, state7, state8)
  c0((x, y, t + 1, newState))
  ???
}
```

But how would the chemistry work at the next time step? The molecule `c0((x, y, t + 1, _))` will need to react with its 8 neighbors. However, each of the neighbor cells, such as `c0((x-1, y, t + 1, _))`, also needs to react with *its* 8 neighbors.

Therefore, we need to have 9 *output* molecules in this reaction: one molecule, `c0()`, represents the new state of the center cell, while 8 others will provide `newState` as "neighbor data" for each of the 8 neighbors. We need to emit these 8 other molecules with shifted coordinates, so that they will react with their proper neighbors at time `t + 1`.

The reaction now looks like this:

```
go { case
  c0((x0, y0, t0, state0)) +
  c1((x1, y1, t1, state1)) +
  c2((x2, y2, t2, state2)) +
  c3((x3, y3, t3, state3)) +
  c4((x4, y4, t4, state4)) +
  c5((x5, y5, t5, state5)) +
  c6((x6, y6, t6, state6)) +
  c7((x7, y7, t7, state7)) +
  c8((x8, y8, t8, state8))
   if x0 == x1 && x0 == x2 && x0 == x3 && x0 == x4 &&
      x0 == x5 && x0 == x6 && x0 == x7 && x0 == x8 &&
      y0 == y1 && y0 == y2 && y0 == y3 && y0 == y4 &&
      y0 == y5 && y0 == y6 && y0 == y7 && y0 == y8 &&
      t0 == t1 && t0 == t2 && t0 == t3 && t0 == t4 &&
      t0 == t5 && t0 == t6 && t0 == t7 && t0 == t8 ⇒
  val newState = getNewState(state0, state1, state2, state3,
    state4, state5, state6, state7, state8)
  c1((x - 1, y - 1, t + 1, newState))
  c2((x + 0, y - 1, t + 1, newState))
  c3((x + 1, y - 1, t + 1, newState))
  c4((x - 1, y + 0, t + 1, newState))

  c0((x + 0, y + 0, t + 1, newState)) // center cell

  c5((x + 1, y + 0, t + 1, newState))
  c6((x - 1, y + 1, t + 1, newState))
  c7((x + 0, y + 1, t + 1, newState))
  c8((x + 1, y + 1, t + 1, newState))
}
```

These reactions work! That's actually a complete chemical program that correctly simulates the Game of Life.

To start the simulation, we need to emit initial molecules. For each cell `(x, y)` on the initial board, we need to emit 9 molecules `c0((x, y, t = 0, _))`, ..., `c8((x, y, t = 0, _))`: `c0()` bearing the initial state and `c1()`, ..., `c8()` providing neighbor data.

Another detail we glossed over is handling the edges of the game board. The simplest solution is to make the board wrap around in both `x` and `y` directions.

The code for emitting the initial molecules with wraparound can be like this:

```
val initBoard: Array[Array[Int]] = ???

(0 until sizeY).foreach { y0 ⇒
 (0 until sizeX).foreach { x0 ⇒
   val initState = initBoard(y0)(x0)
   c0(((x0 + 0 + sizeX) % sizeX, (y0 + 0 + sizeY)) % sizeY), 0, initState))
   c1(((x0 - 1 + sizeX) % sizeX, (y0 - 1 + sizeY)) % sizeY), 0, initState))
   c2(((x0 + 0 + sizeX) % sizeX, (y0 - 1 + sizeY)) % sizeY), 0, initState))
   c3(((x0 + 1 + sizeX) % sizeX, (y0 - 1 + sizeY)) % sizeY), 0, initState))
   c4(((x0 - 1 + sizeX) % sizeX, (y0 + 0 + sizeY)) % sizeY), 0, initState))
   c5(((x0 + 1 + sizeX) % sizeX, (y0 + 0 + sizeY)) % sizeY), 0, initState))
   c6(((x0 - 1 + sizeX) % sizeX, (y0 + 1 + sizeY)) % sizeY), 0, initState))
   c7(((x0 + 0 + sizeX) % sizeX, (y0 + 1 + sizeY)) % sizeY), 0, initState))
   c8(((x0 + 1 + sizeX) % sizeX, (y0 + 1 + sizeY)) % sizeY), 0, initState))
 }
}
```

The complete working code for this implementation is the second test case in `GameOfLifeSpec.scala`.

Improving performance

While the program as written so far works correctly, it works *extremely slowly*. Also, the CPU utilization stays around 100%; in other words, only one CPU core is loaded at 100% while other cores remain idle. Not only the code runs slowly, — it also fails to use any concurrency!

The main reason for the bad performance is the complicated guard condition in the reaction. This guard condition is an example of a **cross-molecule guard**, which means a constraint on the values of a set of molecules as a whole (rather than constraining one molecule value at a time). In our code, the cross-molecule guard constrains the values of all 9 input molecules at once.

Because of the presence of the cross-molecule guard, many combinations of molecule values must be examined before a reaction can be started. Let us make a rough estimate. For an `n * n` board, we initially emit `n * n` copies of molecules for each of the nine sorts `c0`, ..., `c8`. The reaction site needs to find one copy of `c0()`, one copy of `c1()`, etc., such that the guard returns `true` for values carried by these copies. In the worst case, the reaction site will examine `(n * n) ^ 9` possible combinations of molecule values before scheduling a single reaction. There are `n * n` reactions to be scheduled at each time step. This brings the worst-case complexity to `(n * n) ^ 10` per time step. So, even for a

smallest board with `n=2`, we get `(2 * 2) ^ 10 = 1048576` combinations to be examined. A million operations is a very large scheduling overhead for a computation that only runs 4 reactions per time step.

The chemical machine can only go so far in optimizing guard conditions that can contain arbitrary user code. Reactions without guard conditions are scheduled much faster.

How can we rewrite the chemistry so that reactions do not need guard conditions?

We use the cross-molecule guard condition only to select molecules that should react together. In the current code, these molecules are selected by their coordinates in space and time. Instead of using a cross-molecule guard to select input molecules, we can define a *separate reaction* for each group of molecules that react together.

To achieve this, instead of using a single molecule sort `c0` with parameters `(x, y, t, state)`, we will use a new molecule sort for each set of `(x, y, t)`. In other words, we will define *chemically different* molecules representing cells at different `(x, y, t)`. The easiest implementation is by creating a multidimensional matrix of molecule emitters. Now that we are at it, the 9 sorts `c0`, ..., `c9` can be accommodated by an additional dimension in the same matrix; this will save us some boilerplate.

```
val emitterMatrix: Array[Array[Array[Array[M[Int]]]]] =
 Array.tabulate(sizeX, sizeY, sizeT, 9)((x, y, t, label) ⇒
   new M[Int](s"c$label[$x,$y,$t]")
 )
```

The array `emitterMatrix` stores all the molecule emitters we will need. The strings such as `"c8[2,3,0]"`, representing the names of all the new molecules, are assigned explicitly using the `new M()` constructor since the macro `m` would assign the same name to all molecules, which might complicate debugging. Molecule emitters are of type `M[Int]` because the only value that the new molecules need to carry is the integer `state`.

All molecules in `emitterMatrix` are chemically different since they were created using independent calls to `new M()`. It remains to define reactions for these new molecules.

We want to define a separate reaction for each `x`, `y`, `t`, and store all these reactions in an array. Here is code that defines the 3-dimensional array of reactions:

```
val reactionMatrix: Array[Array[Array[Reaction]]] =
 Array.tabulate(boardSize.x, boardSize.y, finalTimeStep) { (x, y, t) ⇒
   // Molecule emitters for the inputs.
   // We need to assign them to separate `val`'s
   // because `case emitterMatrix(x)(y)(t)(0)(state) ⇒ ...` does not compile.
   val c0 = emitterMatrix(x)(y)(t)(0)
   val c1 = emitterMatrix(x)(y)(t)(1)
   val c2 = emitterMatrix(x)(y)(t)(2)
   val c3 = emitterMatrix(x)(y)(t)(3)
   val c4 = emitterMatrix(x)(y)(t)(4)
   val c5 = emitterMatrix(x)(y)(t)(5)
   val c6 = emitterMatrix(x)(y)(t)(6)
   val c7 = emitterMatrix(x)(y)(t)(7)
   val c8 = emitterMatrix(x)(y)(t)(8)

   go { case
       c0(state0) +
       c1(state1) +
       c2(state2) +
       c3(state3) +
       c4(state4) +
       c5(state5) +
       c6(state6) +
       c7(state7) +
       c8(state8) ⇒
     val newState = getNewState(state0, state1, state2, state3,
       state4, state5, state6, state7, state8)

     // Emit output molecules.
     emitterMatrix(x + 0)(y + 0)(t + 1)(0)(newState)
     emitterMatrix(x + 1)(y + 0)(t + 1)(1)(newState)
     emitterMatrix(x - 1)(y + 0)(t + 1)(2)(newState)
     emitterMatrix(x + 0)(y + 1)(t + 1)(3)(newState)
     emitterMatrix(x + 1)(y + 1)(t + 1)(4)(newState)
     emitterMatrix(x - 1)(y + 1)(t + 1)(5)(newState)
     emitterMatrix(x + 0)(y - 1)(t + 1)(6)(newState)
     emitterMatrix(x + 1)(y - 1)(t + 1)(7)(newState)
     emitterMatrix(x - 1)(y - 1)(t + 1)(8)(newState)
   }
}
```

Now that all reactions are created, we need to define a reaction site that will run them. The `site()` call accepts a sequence of reactions. Since `reactionMatrix` is a 3-dimensional array, we need to flatten it twice:

```
site(reactionMatrix.flatten.flatten)
```

This implementation is found as test 4 in `GameOfLifeSpec.scala`. This runs 10 time steps on a 10x10 game board in about the same time as the previous implementation ran 1 time step on a 2x2 board.

Radically improving performance

We have greatly increased the speed of the simulation, but there is still room for improvement. The CPU usage of the new solution is still around 100% much of the time, which indicates that parallelism is not optimal. The reason for low parallelism is the low granularity of the reaction sites: we have a single reaction site that runs all reactions. A single reaction site can usually schedule only one reaction at a time, and our reactions are very simple, so we still fail to take advantage of multicore parallelism.

To make progress, we notice that the program defines a separate reaction for each cell, and each reaction has its own chemically unique input molecules. In this case, we can define *each reaction* at a separate reaction site, instead of defining all reactions within one reaction site.

The only code change we need to make is the definition of the reaction sites, which will now look like this:

```
reactionMatrix.foreach(_.foreach(_.foreach(r ⇒ site(r)())))
```

This implementation is found as test 6 in `GameOfLifeSpec.scala`. It runs about 200 times faster than the previous implementation (test 4): it computes 100 time steps on a 10x10 board in about 0.8 seconds.

Since each reaction is running at a separate reaction site, every cell update can be scheduled concurrently with any other cell update. With this implementation, cell updates can be concurrent across the entire 3-dimensional array of cells (that is, both in space and time).

Conclusion

We have seen that there are two ways of optimizing the performance of a chemical computation:

1. Redesign the chemistry so that reactions do not need cross-molecule guard conditions.
2. Redesign the chemistry so that many independent reaction sites are used, with fewer reactions per reaction site.

To see the effect of these design choices, and to provide a benchmark for the chemical machine, I made six different implementations of the Game of Life that differ only in the design of reactions.

The implementation in test 1 uses a single reaction with a single molecule sort. The reaction has 9 repeated input molecules and emits 9 copies of the same molecule. All coordination is performed by the guard condition that selects input molecules for reactions.

Test 2 is the solution first discussed in this chapter. It introduces 9 different molecule sorts `c0`, ..., `c8` instead of using one molecule sort. Otherwise, the chemistry remains the same as in test 1. The change speeds up the simulation by a few times, although it remains unacceptably slow.

Tests 1 and 2 are intentionally very slow, to be used as benchmarks of the chemical machine. The speedup between 1 and 2 suggests that avoiding repeated input molecules is a source of additional speedup. This may or may not remain the case in future versions of `Chymyst`.

Test 3 uses a different molecule sort for each cell on the board. However, molecules corresponding to different time steps are the same. The reaction still needs a cross-molecule guard condition to match up input molecules at the same time step. Using different molecules for different cells speeds up the simulation enormously.

Tests 4—6 use a different molecule sort for each cell on the board and for each time step. This is the solution discussed in the later sections of this chapter. The reactions need no guard conditions, which results in a significant speedup.

The difference between tests 4—6 is in the granularity of reaction sites. Test 4 has all reactions in one reaction site; test 5 declares a new reaction site for each time step; test 6 declares a new reaction site for each cell and for each time step. The speedup between tests 4 and 6 is about 20x.

The complete working code showing the six different implementations is found in `GameOfLifeSpec.scala`.

From actors to reactions: The chemical machine explained through the Actor model

Many Scala developers interested in concurrent programming are already familiar with the Actor model. In this brief chapter, I outline how the chemical machine paradigm can be introduced to those readers.

In the Actor model, an actor receives messages and reacts to them by running a computation. An actor-based program declares several actors, defines the computations for them, stores references to the actors, and starts sending messages to some of the actors. Messages are sent either synchronously or asynchronously, enabling communication between different concurrent actors.

The chemical machine paradigm is in certain ways similar to the Actor model. A chemical program also consists of concurrent processes, or “chemical actors”, that communicate by sending messages. The chemical machine paradigm departs from the Actor model in two major ways:

1. Chemical actors are automatically started and stopped; the user's code only sends messages and does not manipulate actor references.
2. Chemical actors may wait for a set of different messages to be received atomically.

If we examine these requirements and determine what should logically follow from them, we will arrive at the chemical machine paradigm.

The first requirement means that chemical actors are not created explicitly by the user's program. Instead, the chemical machine runtime will automatically instantiate and run a chemical actor whenever some process sends a relevant input message. A chemical actor will be automatically stopped and deleted when its computation is finished. Therefore, the user's code now does not create an instance of an actor but merely *defines the computation* that an auto-created actor will perform after consuming a message. As a consequence, a chemical actor must be *stateless* and only perform computations that are functions of the input message values.

Implementing this functionality will allow us to write pseudo-code like this,

```
val c1 = go { x: Int ⇒ ... }
c1 ! 123
```

The computation under the `go()` keyword receives a message with an `Int` value and performs some processing on it. The computation will be instantiated and run concurrently, whenever a message is sent. In this way, we made the first step towards the chemical machine paradigm.

What should happen if we quickly send many messages?

```
val c1 = go { x: Int ⇒ ... }
(1 to 100).foreach { c1 ! _ }
```

Since our computations are stateless, it is safe to run several instances of the computation `{ x: Int ⇒ ... }` concurrently. The runtime engine may automatically adjust the degree of parallelism depending on CPU load.

Note that `c1` is not a reference to a particular *instance* of the computation. Rather, the computation `{ x: Int ⇒ ... }` is merely a declarative description of what needs to be done with any message sent via `c1` . We could say that the value `c1` plays the role of a *label* attached to the value `123` . This label implies that the value `123` should be used as the input parameter `x` in a particular computation.

To express this semantics more clearly, let us change our pseudo-code notation to

```
go { x: Int from c1 ⇒ ... }
c1 ! 123
```

Different chemical actors are now distinguished only by their input message labels, for example:

```
go { x: Int from c1 ⇒ ... }
go { x: Int from d1 ⇒ ... }
c1 ! 123
d1 ! 456
```

Actor references have disappeared from the code. Instead, input message labels such as `c1` , `d1` select the computation that will be started.

The second requirement means that a chemical actor should be able to wait for, say, two messages at once, allowing us to write pseudo-code like this,

```
go { x: Int from c1, y: String from c2 ⇒ ... }
c1 ! 123
c2 ! "abc"
```

The two messages carry data of different types and are labeled by `c1` and `c2` respectively. The computation starts only after *both* messages have been sent, and consumes both messages atomically.

It follows that messages cannot be sent to a linearly ordered queue or a mailbox. Instead, messages must be kept in an unordered bag, as they will be consumed in an unknown order.

It also follows from the atomicity requirement that we may define several computations that *jointly contend* on input messages:

```
go { x: Int from c1, y: String from c2 ⇒ ... }
go { x: Int from c1, z: Unit from e1 ⇒ ... }
```

Messages that carry data are now completely decoupled from computations that consume the data. All computations start concurrently whenever their input messages become available. The runtime engine needs to resolve message contention by making a non-deterministic choice of the messages that will be actually consumed.

This concludes the second and final step towards the chemical machine paradigm where "chemical actors" are called **reactions**, "messages" are **molecules**, and "input message labels" are **molecule emitters**.

It remains to use the Scala syntax instead of pseudo-code. In Scala, we need to declare message types explicitly and to register reactions with the runtime engine as a separate step. The syntax used by `Chymyst` looks like this:

```
val c1 = m[Int]
val c2 = m[String]
site(go { c1(x) + c2(y) ⇒ ... })
c1(123)
c2("abc")
```

Here, `m[Int]` creates a new message label with values of type `Int`.

As we have just seen, the chemical machine paradigm is a radical departure from the Actor model.

Whenever there are sufficiently many input messages available for processing, the runtime engine may automatically instantiate several concurrent copies of the same reaction that will consume the input messages concurrently. This is the main method for achieving parallelism in the chemical paradigm. The runtime engine is in the best position to balance the CPU load over low-level threads. The application code does not need to specify how many parallel processes to run at any given time.

Since reactions are stateless and instantiated automatically on demand, the application code does not need to manipulate explicit actor references, which is error-prone. (For example, books on Akka routinely warn against capturing `sender()` in a `Future`, which may yield an incorrect actor reference when the `Future` is resolved.) The application code also does not need to implement actor lifecycle management, actor hierarchies, backup and recovery of actors' internal state, or dead with the special “dead letter” actor. This removes a significant amount of complexity from the architecture of concurrent applications.

Input message contention is used in the chemical machine paradigm as a general mechanism for synchronization and mutual exclusion. (In the Actor model, these features are implemented by creating a fixed number of actor instances that alone can consume certain messages.) Since the runtime engine will arbitrarily decide which actor to run, input contention will result in indeterminism. This is quite similar to the indeterminism in the usual models of concurrent programming. For example, mutual exclusion allows the programmer to implement safe exclusive access to a resource for any number of concurrent processes, but the order of access among the contending processes remains unspecified.

Other work on Join Calculus

Here are other implementations of Join Calculus that I was able to find online.

- The `Funnel` programming language: M. Odersky et al., 2000. This project was discontinued, and Odersky went on to create the Scala language, which does not include any concepts from Funnel or JC.
- *Join Java*: von Itzstein et al., 2001-2005. This was a modified Java language compiler, with support for certain Join Calculus constructions. The project is not maintained.
- The `JoCaml` language: Official site and a publication about JoCaml: Fournet et al. 2003. This project embeds JC into OCaml and is implemented as a patch to the mainstream OCaml compiler. The project is still maintained, and a full JoCaml distribution is available with the OCaml OPAM platform.
- "Join in Scala" compiler patch: V. Cremet 2003. The project is discontinued.
- `Joins` library for .NET: P. Crusso 2006. The project is available as a .NET binary download from Microsoft Research, and is not maintained.
- `ScalaJoins`, a prototype implementation in Scala: P. Haller 2008. The project is not maintained.
- `ScalaJoin`, an improvement over `ScalaJoins`: J. He 2011. The project is not maintained.
- "Joinads", a Join-Calculus implementation as a compiler patch for F# and Haskell: Petricek and Syme 2011. The project is not maintained.
- Implementations of Join Calculus for iOS: CocoaJoin and for Android: AndroJoin. These projects are not maintained.
- Join-Language: implementation of Join Calculus as an embedded Haskell DSL (2014). The project is in development.

The implementation of JC in `Chymyst` is based on ideas from Jiansen He's `ScalaJoin` as well as on CocoaJoin / AndroJoin.

Improvements with respect to Jiansen He's `ScalaJoin`

Compared to `ScalaJoin` (Jiansen He's 2011 implementation of JC), `Chymyst Core` offers the following improvements:

- Lighter syntax for reaction sites ("join definitions"). Compare:

`Chymyst Core`:

```
val a = m[Int]
val c = b[Int, Int]
site(
  go { case a(x) + c(y, reply) ⇒
    a(x + y)
    reply(x)
  }
)
a(1)
```

`ScalaJoin`:

```
object join1 extends Join {
  object a extends AsyName[Int]
  object c extends SynName[Int, Int]

  join {
    case a(x) and c(y) ⇒
      a(x + y)
      c.reply(x)
  }
}
a(1)
```

- Molecule emitters ("channel names") are not singleton objects as in `ScalaJoin` but locally scoped values. This is how the semantics of JC is implemented in JoCaml. In this way, we get more flexibility in defining molecules.
- Reactions are not merely `case` clauses but locally scoped values of type `Reaction`. `Chymyst` uses macros to perform static analysis of reactions at compile time and detect some errors.
- Reaction sites are not static objects — they are local values of type `ReactionSite` and are invisible to the user, as they should be according to the semantics of JC.

Improvements with respect to JoCaml

As a baseline reference, the most concise syntax for JC is available in JoCaml, which uses a modified OCaml compiler:

```
def a(x) & c(y) =
   a(x+y) & reply x to c
spawn a(1)
```

In the JoCaml syntax, `a` and `c` are declared implicitly, together with the reaction. Implicit declaration of molecule emitters ("channels") is not possible in `Chymyst` because Scala macros do not allow us to insert a new top-level name declaration into the code. So, molecule declarations need to be explicit and show the types of values (`b[Int, Int]` and so on). Other than that, `Chymyst` 's syntax is closely modeled on that of `ScalaJoin` and JoCaml.

Another departure from JoCaml is that the linearity restriction is lifted for input patterns. In `Chymyst` , reactions can declare any number of repeated input molecules, as well as repeated blocking molecules. The novel reply syntax disambiguates the destination of the reply value:

```
go { case a(x1, reply1) + a(x2, reply2) ⇒ reply2(x1); reply1(x2) }
```

This reaction is impossible to express in JoCaml directly.

Other tutorials on Join Calculus

This tutorial is a pragmatic, non-theoretical introduction to Join Calculus for programmers.

This text is based on my earlier tutorial for JoCaml. (However, be warned that the old JoCaml tutorial is unfinished and contains mistakes in some of the more advanced code examples.)

See also my recent presentation at *Scalæ by the Bay 2016*. (Talk slides are available). That presentation covered an early version of `Chymyst` .

There are a few academic papers on Join Calculus and a few expository descriptions, such as the Wikipedia article or the JoCaml documentation. Unfortunately, I cannot recommend reading them because they are unsuitable for learning about the chemical machine / Join Calculus paradigm.

I learned about the "Reflexive Chemical Abstract Machine" from the introduction in one of the early papers on Join Calculus.

Do not start by reading academic papers if you never studied Join Calculus - you will be unnecessarily confused. All Join Calculus papers I've seen are not pedagogically written and are intended for advanced computer scientists.

As another comparison, here is some code taken from this tutorial, written in academic Join Calculus notation:

$$\begin{array}{l}\texttt{def}\ newVar(v_0) \triangleright \\ \quad \texttt{def}\ put(w) \mid val\langle v\rangle \triangleright val\langle w\rangle \mid \texttt{return} \\ \quad\quad \wedge\ get() \mid val\langle v\rangle \triangleright val\langle v\rangle \mid \texttt{return}\ v\ \texttt{in} \\ \quad val\langle v_0\rangle \mid \texttt{return}\ put, get\ \texttt{in}\ \ldots\end{array}$$

This code creates a shared value container `val` with synchronized single access.

The equivalent `Chymyst` code looks like this:

```
def newVar[T](v0: T): (B[T, Unit], B[Unit, T]) = {
  val put = b[T, Unit]
  val get = b[Unit, T]
  val vl = m[T] // have to use `vl` since `val` is a Scala keyword

  site(
    go { case put(w, ret) + vl(v) ⇒ vl(w); ret() },
    go { case get(_, ret) + vl(v) ⇒ vl(v); ret(v) }
  )
  vl(v0)

  (put, get)
}
```

I also do not recommend reading the Wikipedia page on Join Calculus. As of June 2017, this page says this about Join Calculus:

> The join-calculus ... can be considered, at its core, an asynchronous π-calculus with several strong restrictions:
>
> - Scope restriction, reception, and replicated reception are syntactically merged into a single construct, the *definition*;
> - Communication occurs only on defined names;
> - For every defined name there is exactly one replicated reception.
>
> However, as a language for programming, the join-calculus offers at least one convenience over the π-calculus — namely the use of multi-way join patterns, the ability to match against messages from multiple channels simultaneously.

This explanation is impossible to understand unless you are already well-versed in the research literature. (I'm not sure what it means to have “communication on *defined* names”, as opposed to communication on *undefined* names...)

Academic literature on Join Calculus typically uses terms such as “channel” and “message”, which are not helpful for understanding how JC works and how to write concurrent programs in JC. Indeed, a “channel” in JC holds an *unordered* collection of messages, rather than an

ordered queue or mailbox, as the word “channel” suggests. Another metaphor for “channel” is a persistent path for sending and receiving messages, but this is also far from what a JC “channel” actually does.

The word “message” again suggests that a mailbox or a queue receives messages and processes them one by one. This is very different from what happens in JC, where a reaction waits for several “messages” at once, and different reactions can contend on several “messages” they wait for.

The JoCaml documentation is especially confusing as regards “channels”, “messages”, and “processes”. It is ill-suited as a pedagogical introduction to using JoCaml.

Instead of using academic terminology, I always follow the chemical machine metaphor and terminology when talking about `Chymyst` programming. Here is a dictionary:

Chemical machine	Academic Join Calculus	`Chymyst` code
input molecule	message on a channel	`case a(123) ⇒ ...` *// pattern-matching*
molecule emitter	channel name	`val a: M[Int]`
blocking emitter	synchronous channel	`val q : B[Unit, Int]`
reaction	process	`val r1 = go { case a(x) + ... ⇒ ... }`
emitting a molecule	sending a message	`a(123)` *// side effect*
emitting a blocking molecule	sending a synchronous message	`q()` *// returns* `Int`
reaction site	join definition	`site(r1, r2, ...)`