Reactamole: Functional Reactive Molecular Programming

Titus H. Klinge
Drake University, Des Moines, IA, USA

James I. Lathrop
Iowa State University, Ames, IA, USA

Peter-Michael Osera
Grinnell College, Grinnell, IA, USA

Allison Rogers
Grinnell College, Grinnell, IA, USA

Abstract

Chemical reaction networks (CRNs) are an important tool for molecular programming, a field that is rapidly expanding our ability to deploy computer programs into biological systems for a variety of applications. However, CRNs are also difficult to work with due to their massively parallel nature, leading to the need for higher-level languages that allow for easier computation with CRNs. Recently, research has been conducted into a variety of higher-level languages for deterministic CRNs but modeling CRN parallelism, managing error accumulation, and finding natural CRN representations are ongoing challenges.

We introduce Reactamole, a higher-level language for deterministic CRNs that utilizes the functional reactive programming (FRP) paradigm to represent CRNs as a reactive dataflow network. Reactamole equates a CRN with a functional reactive program, implementing the key primitives of the FRP paradigm directly as CRNs. The functional nature of Reactamole makes reasoning about molecular programs easier, and its strong static typing allows us to ensure that a CRN is well-formed by virtue of being well-typed. In this paper, we describe the design of Reactamole and how we use CRNs to represent the common datatypes and operations found in FRP. We also demonstrate the potential of this functional reactive approach to molecular programming by giving an extended example where a CRN is constructed using FRP to modulate and demodulate an amplitude modulated signal.

2012 ACM Subject Classification Software and its engineering → Functional languages; Software and its engineering → Data flow languages

Keywords and phrases Chemical Reaction Network, Functional Reactive Programming, Domain Specific Language

Digital Object Identifier 10.4230/LIPIcs.DNA.27.10

Supplementary Material *Software (Source Code)*: `https://github.com/digMP/haskell-reactamole` archived at `swh:1:dir:9f8a912be45038fedf03c9dd6bf74a6a20f29aff`

Funding *James I. Lathrop*: This research was supported in part by NSF grants 1545028, 1900716, and 1909688.
Peter-Michael Osera: This research was supported in part by NSF grants 1651817 and 2049911.

Acknowledgements We thank the four anonymous reviewers for their feedback on this paper and Noah Susag for his contributions to the Reactamole code.

1 Introduction

Molecular programming harnesses computer science towards designing programmable structures at the nanoscale, unlocking the potential to execute programs in biological systems. This emerging arena holds significant potential for innovations in medicine, nanofabrication, and synthetic biology. One prominent molecular programming language is chemical reaction

© Titus H. Klinge, James I. Lathrop, Peter-Michael Osera, and Allison Rogers;
licensed under Creative Commons License CC-BY 4.0

networks (CRNs), abstractions of chemical reactions [8, 5, 10]. CRNs are Turing-complete and act as an unstructured assembly language for molecular programming, which can then be assembled into DNA to perform computation at the nanoscale [9, 1].

However, the characteristics of CRNs create substantial challenges for general programmability. Due to the nature of chemical reactions, CRNs are massively parallel, with all reactions active at the same time depending on the availability of the reactants. This creates race conditions that can make coding in this framework more difficult and error-prone [22]. CRNs are also unstructured and not easily composed; adding a single reaction can easily have global side effects to its behavior. This is strong motivation for the creation of a high-level programming language to abstract away these barriers.

Consequently, recent research has been conducted into high-level languages for molecular programming such as CRN++ [22] and Kaemika [4]. CRN++ enables programming in a familiar, imperative style that compiles to deterministic CRNs (DCRNs), marking a significant advancement in this realm but also leaving room for continued improvement. Vasic et al. say that CRN++ could be improved by addressing inefficiencies caused by careful avoidance of the inherent parallelism of CRNs, as well as reducing errors accumulated over time [22]. Kaemika supports specifying CRNs in a functional style, including support for high-order functions and recursion [4]. Both CRN++ and Kaemika allow side effects, with Kaemika depending on these side effects for the generation of new species within functions. Additionally, these implementations leave room to improve the synchronicity between language structure and CRN behavior.

We address these downsides by exploring the use of *functional reactive programming* (*FRP*) for developing CRNs. Functional reactive programming is a paradigm primarily characterized by its reactive nature, responding to stimuli through continuous and discrete time-dependent inputs [2]. In FRP, systems are modeled as graphs where nodes are operations and edges indicate how data flows between these operations, with a particular focus on how change is propagated through this graph. We observe a close correspondence between CRNs and functional reactive programs. The chemical concentrations of a CRN react to changes in their environment similarly and can thus be thought of as *signals*, time-varying data streams, in a functional reactive program. Consequently, CRNs themselves transform these concentrations and can, therefore, be thought of as functions over signals, i.e., *signal functions*. These correspondences make FRP a natural choice to express computation within a CRN.

We use this correspondence to design a *functional reactive molecular programming* (FRMP) language for deterministic CRNs. The heart of this language is the expression of the core constructs of FRP directly in terms of CRNs, gaining the benefits of program composability afforded by the functional reactive paradigm. We explore this approach to molecular programming with REACTAMOLE, an embedded domain specific language (eDSL) for FRMP modeled after Yampa, a prominent functional reactive programming DSL [14]. Furthermore, by combining FRP and CRNs, we open the door for applying recent advancements in programming language theory towards the development of CRNs. For example, in this FRMP paradigm, we can now consider integrating type systems that verify safety properties of functional reactive programs [18] or program synthesizers for functional reactive programs [11].

In Section 2, we review the basic definitions of chemical reaction networks as well as introduce the necessary components of the Haskell programming language and functional reactive programming needed to understand REACTAMOLE. We introduce REACTAMOLE by way of example in Section 3 and describe the design and implementation of REACTAMOLE in Section 4. Finally, we demonstrate the potential of REACTAMOLE by way of a case study – implementing amplitude modulation over real-valued signals – in Section 5.

2 Background

In this section, we review the two main topics that we combine in REACTAMOLE: chemical reactions networks (Subsection 2.1) and functional reactive programming (Subsection 2.2). Throughout the remaining sections, we assume familiarity with the basic syntax and semantics of the Haskell programming language. We provide a walkthrough of the parts of Haskell necessary to understand our presentation in Appendix A.

2.1 Chemical Reaction Networks

Scientists and researchers often utilize models to describe the complex interactions of molecules and matter. These models simplify molecular interactions in order to allow algorithms and software to simulate the outcomes of chemical systems. Even though these models make simplifying assumptions, they are still a powerful tool for designing and testing these systems. A chemical reaction network (CRN) is one such model that is often used to model molecule interactions in a well-mixed solution. Even though assumptions are made in each of the many variants of the CRN model, almost all of them retain the power to facilitate general computing.

We adopt much of the notation used by Klinge, Lathrop, and Lutz [20] to formalize the CRN model used here. A CRN is a pair $N = (S, R)$, where S is a set of species (molecules) and R is a set of reactions that operate over those species. Reactions are a triple $\rho = (\mathbf{r}, \mathbf{p}, k)$, where $\mathbf{r} \in \mathbb{N}^S$, $\mathbf{p} \in \mathbb{N}^S$, and $k \in (0, \infty)$. Note that $\mathbb{N}^S$ is the set of functions mapping species to non-negative integers, and we do not allow $\mathbf{r} = \mathbf{p}$. The constant k represents the rate constant for the reaction. In this paper we usually represent CRNs simply as a set of reactions that implicitly define a set of species. For example, the reaction

$$X + U \xrightarrow{k} 2X$$

can be interpreted as the CRN $N = (S, R)$ with $S = \{X, U\}$ and R consisting of only the given reaction.

In this paper, we focus on the deterministic mass-action model of chemical reaction networks, where species are represented by concentrations of molecules. This is in contrast to the stochastic mass-action model, which uses the *number* of molecules for this purpose. The deterministic mass-action model describes interactions with molecules using polynomial autonomous differential equations, and the semantics relate concentrations of species through the reactants, products, and rate constants of all reactions in the system. The single reaction above yields the set of ordinary differential equations (ODEs):

$$\frac{dx}{dt} = x \cdot u \cdot k \qquad\qquad \frac{du}{dt} = -x \cdot u \cdot k.$$

Intuitively, since an X and a U react together to produce an additional X in the system, the instantaneous rate of X gained is proportional to the product of the instantaneous amounts of the reactants and the rate constant. Simultaneously, the loss of U occurs at the same rate. Note that we use $x(t)$ or x to denote the concentration of species X.

Similar to Klinge, Lathrop, and Lutz [20], we define an *input/output CRN* (I/O CRN) as a tuple $N = (S, R, I, O)$, where S is a set of species, R is a set of reactions, $I \subseteq S$ is the set of input species, and $O \subseteq S$ is the set of output species. I/O CRNs require that the input species are only used as a catalyst in any reaction, a critical feature that allows us to compose CRNs in REACTAMOLE safely.

2.2 Functional Reactive Programming

Many classes of computation can be expressed as programs that propagate change in response to external stimuli. For example, with:

- Graphical user interfaces (GUIs), interface elements update in response to user input, e.g., mouse movement.
- Spreadsheets, cells that are related via (potential cyclic) references update whenever the user modifies their contents.
- Circuits, input signals propagate through interconnected electrical components.

We could model these phenomena with mutable state. The resulting program would then bear the responsibility of orchestrating how the different pieces of state change in response to the outside world. However, by doing so, we would lose the benefits of composability, a hallmark of the functional style of programming [17].

Functional Reactive Programming (FRP) [6] purifies this stateful situation by modeling values that react with the outside world as *signals*:

```
type Signal a = Time -> a
```

That is, a signal of some arbitrary type `a` is a time-varying value, i.e., a function from time to `a`. For example, an electrical pulse that we might measure using two states, on and off, could be represented as a `Bool`. In an FRP setting, this pulse would be represented as a type, `Signal Bool`, a function describing how the pulse changes over time.

Within FRP, there are a multitude of approaches and variations to address implementation concerns such as space efficiency or design constraints such as modeling discrete versus continuous time and static versus dynamic dependencies between components. In this work, we focus on *arrowized FRP*, which uses the arrow abstraction of Hughes [16], a generalization of composable computation. These arrows take the form of *signal functions* in arrowized FRP, i.e., transformers over signals:

```
type SF a b = Signal a -> Signal b
```

For example, a function `not` that inverts an electrical pulse would have the type `SF Bool Bool`, a signal function that takes a Boolean signal as input and produces a Boolean signal as output.

Some of these signal functions, like `not`, transform our time-varying values directly. Other signal functions are *higher-order* signal functions which take other signal functions as input and produce them as output. These signal function *combinators* allow us to build up more complex signal functions from simpler ones. The most common of these is function composition, traditionally written in the arrow style as the binary operator `(>>>)`:

```
(>>>) :: SF a b -> SF b c -> SF a c
```

`f >>> g` is the composition of signal functions `f` and `g` where the output of `f` (of type `b`) is fed into `g` as input. The resulting signal function takes an input for `f` (of type `a`) and produces an output from `g` (of type `c`) as its result. Other common signal function combinators that we will use in the subsequent sections include:

- The *split operator* `(***) :: SF a b -> SF c d -> SF (a, c) (b, d)` which takes two input signal functions f and g and creates a signal function whose inputs and outputs are *pairs* drawn from f and g.
- The *fanout operator* `(&&&) :: SF a b -> SF a c -> SF a (b, c)` which takes two input signal functions f and g that take a common input type `a` and creates a new signal function that pipes its input independently through both f and g and produces their outputs as a pair.

3 Introducing Reactamole

In this section, we give a brief summary of REACTAMOLE and its uses. Note that many of the REACTAMOLE primitives discussed in this section are further explained later in the paper.

Klinge, Lathrop, and Lutz [20] regarded a chemical reaction network as a device that transforms an input signal into an output signal. The authors defined an *input/output chemical reaction network* (*I/O CRN*) to be a CRN with some species reserved as "inputs" that can only be used catalytically and some species labeled "outputs." These I/O CRNs are literally signal functions in the functional reactive programming sense. As a result, I/O CRNs are specified using the arrow combinators that we introduced in Subsection 2.2. Below, we demonstrate the expressive power of these arrowized I/O CRNs.

We begin with I/O CRNs that produce real numbers. Since species concentrations are non-negative, we encode a real number as the difference between two species. (This is a common technique and was used to show that CRNs are equivalent to the general purpose analog computer (GPAC) [13]). Thus, a real-valued signal $x(t)$ is encoded as $x^+(t) - x^-(t)$ where X^+ and X^- are two species.

One of the simplest I/O CRN operations is the *integrator*. An integrator takes a real-valued signal $x(t)$ and produces the real-valued output signal $y(t) = \int_0^t x(s)ds + y_0$. In REACTAMOLE, we provide an integrator as an I/O CRN primitive:

```
integrate :: Double -> CRN Double Double
```

`integrate` is a function that takes a real-valued parameter `y0` and returns a signal function that performs integration. The parameter `y0` corresponds to the constant $y(0)$ in the solution to $y(t)$. We implement the `integrate` function, with parameter `y0`, using an I/O CRN consisting of the three reactions:

$$X^+ \xrightarrow{1} X^+ + Y^+ \qquad X^- \xrightarrow{1} X^- + Y^- \qquad Y^+ + Y^- \xrightarrow{1} \emptyset$$

The third reaction is an *annihilation* reaction that has no effect on the encoded value of $y(t)$ but ensures that concentrations of Y^+ and Y^- remain close to zero. We can verify the correctness of the above CRN by examining the induced ODEs according to the law of mass action:

$$\frac{dy^+}{dt} = x^+ - y^+y^-, \qquad \frac{dy^-}{dt} = x^- - y^+y^-.$$

Since the functions are dual-rail, we know that $x(t) = x^+(t) - x^-(t)$ and $y(t) = y^+(t) - y^-(t)$. Thus, we can rewrite the above ODEs in terms of x and y directly:

$$\frac{dy}{dt} = \frac{dy^+}{dt} - \frac{dy^-}{dt} = x^+ - x^- = x.$$

By integrating both sides, we obtain the solution $y(t) = \int_0^t x(s)ds + y_0$.

Other primitives we implement in REACTAMOLE include the following.

- `neg :: CRN Double Double` performs numerical negation by reversing the roles of X^+ and X^-. Note that this primitive does not generate additional reactions. Instead, it simply *reinterprets* the output species of the CRN.
- `id :: CRN a a`, the identity signal function, produces its inputs as outputs without modification.
- `proj1 :: CRN (a, b) a` and `proj2 :: CRN (a, b) b` take *pairs* of signals as inputs and project out the individual components of those pairs as output.

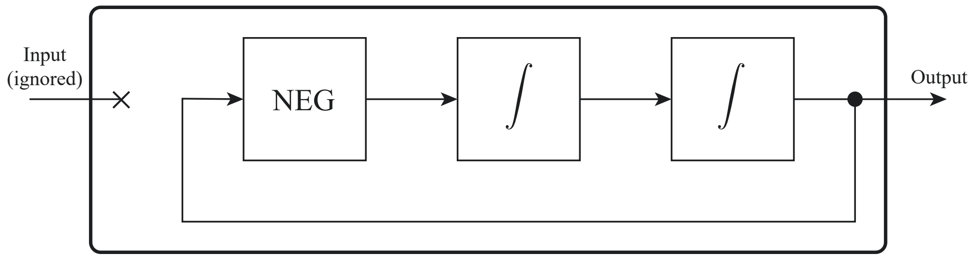

Figure 1 Visual representation of the arrowized `sin` implementation.

- `dup :: CRN a (a, a)` produces a pair where each component is a "copy" of the input. However, in reality, we don't make a copy of the input species. Multiple consuming CRNs can use the components of the resulting pair because I/O CRNs are catalytic in their inputs.
- `loop :: CRN (a, c) (b, c) -> CRN a b` creates a *feedback* loop where the second component of the output of the CRN is given to itself as input.

Using these primitives, it is already possible to specify complex signals such as sine and cosine. We can define `sin` in the following way which is also illustrated in Figure 1:

```
sin :: CRN a Double
sin = loop (proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup)
```

This definition exploits the fact that the sine function satisfies the second-order differential equation $x''(t) = -x(t)$. Since the output of `sin` does not depend on its input, the type signature of `sin` has an abstract input type `a`, meaning that it can receive any input. The `loop` combinator creates a feedback loop that allows the signal $x(t)$ to depend on itself. Since `sin` uses two applications of `integrate`, Reactamole generates the following six reactions:

$$\begin{array}{lll} X_1^+ \xrightarrow{1} X_1^+ + X_2^- & X_2^+ \xrightarrow{1} X_2^+ + X_1^+ & X_1^+ + X_1^- \xrightarrow{1} \emptyset \\ X_1^- \xrightarrow{1} X_1^- + X_2^+ & X_2^- \xrightarrow{1} X_2^- + X_1^- & X_2^+ + X_2^- \xrightarrow{1} \emptyset \end{array}$$

We can verify the correctness of the `sin` definition by observing that the ODEs associated with the reactions satisfy

$$\frac{dx_1}{dt} = \frac{dx_1^+}{dt} - \frac{dx_1^-}{dt} = x_2^+ - x_2^- = x_2 \qquad \frac{dx_2}{dt} = \frac{dx_2^+}{dt} - \frac{dx_2^-}{dt} = x_1^- - x_1^+ = -x_1$$

and therefore $x_1(t) = \sin(t)$ and $x_2(t) = \cos(t)$.

We now turn our attention to I/O CRNs that produce Booleans. Similar to real numbers, we encode a Boolean signal using a pair of species $(X, \overline{X})$ while maintaining the invariant that $x(t) + \overline{x}(t) = 1$. When species X is high, the Boolean is interpreted as *true*; similarly, when $\overline{X}$ is high, the Boolean is interpreted as *false*.

The elementary Boolean signal functions in Reactamole are `not :: CRN Bool Bool` and `nand :: CRN (Bool, Bool) Bool`. Similar to `neg`, the `not` signal function "crosses the wires" of X and $\overline{X}$ without adding any additional species or reactions. The `nand` signal function consists of five reactions, using the implementation provided by Ellis, Klinge, and Lathrop [7] and visualized in Figure 2.

Using these two primitives, we can define the other elementary gates as follows.

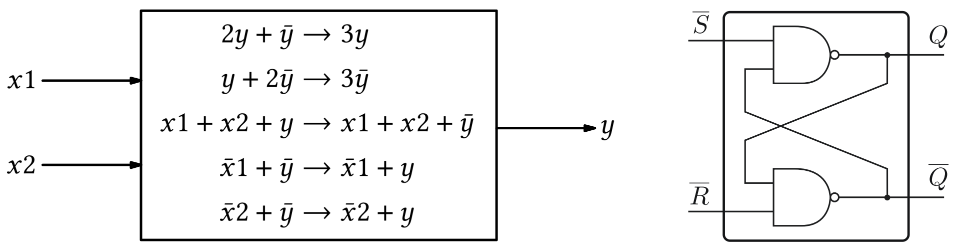
$2y + \bar{y} \longrightarrow 3y$
$y + 2\bar{y} \longrightarrow 3\bar{y}$
$x1 + x2 + y \longrightarrow x1 + x2 + \bar{y}$
$\bar{x}1 + \bar{y} \longrightarrow \bar{x}1 + y$
$\bar{x}2 + \bar{y} \longrightarrow \bar{x}2 + y$
$x1$
$x2$
y
$\overline{S}$
$\overline{R}$
Q
$\overline{Q}$

Figure 2 Reactamole `nand` and `srLatch` implementations.

```
or, and, nor, xor, xnor :: CRN (Bool, Bool) Bool
or  = (not *** not) >>> nand
and = nand >>> not
nor = or >>> not
xor = (nand &&& or) >>> and
xnor = xor >>> not
```

Note that `not` does not introduce additional overhead, so the signal functions `or`, `and` and `nor` are no more complex than `nand`.

We can also use the `loop` combinator to create sequential logic gates such as a *set-reset latch* which is visualized in Figure 2.

```
srLatch :: CRN (Bool, Bool) (Bool, Bool)
srLatch = loop (crossWires >>> (nand *** nand) >>> dup)
```

Here, `crossWires` is a simple signal function that “rearranges the wires” so that the outputs are looped back into the appropriate `nand` gates. Since `srLatch` has two Boolean inputs and is implemented with two `nand` gates, the resulting I/O CRN consists of eight species and nine reactions[1].

Finally, we can create signal functions that employ both real-valued signals and Boolean signals. For example, we include a primitive `isPos :: CRN Double Bool` that tests if a real-valued input is positive. Note that `isPos` is a continuous approximation of a discontinuous function, so its Boolean output is undefined if the input signal is close to zero.

Using `isPos` and the previously defined `sin` signal, we can easily create a `clock` signal that could be employed in clocked sequential circuits.

```
clock :: CRN a Bool
clock = sin >>> isPos
```

[1] Reactamole automatically optimizes the final I/O CRN by combining reactions together when possible. Thus, `srLatch` consists of nine reactions instead of ten because two reactions can be safely combined into one without affecting the underlying ODEs of the CRN.

```
-- Algebraic structures
(>>>)  :: CRN a b -> CRN b c -> CRN a c
(***)  :: CRN a b -> CRN c d -> CRN (a, c) (b, d)
(&&&)  :: CRN a b -> CRN a c -> CRN a (b, c)
first  :: CRN a b -> CRN (a, c) (b, c)
second :: CRN a b -> CRN (c, a) (c, b)
-- Booleans
not    :: CRN Bool Bool
nand   :: CRN (Bool, Bool) Bool
arr1Bl :: (Bool -> Bool) -> CRN Bool Bool
-- Reals
integrate :: Double -> CRN Double Double
neg       :: CRN Double Double
add       :: CRN (Double, Double) Double
mult      :: CRN (Double, Double) Double
isPos     :: CRN Double Bool
```

```
-- Switching
(+++) :: CRN a b -> CRN c d
      -> CRN (Either a c)
             (Either b d)
(|||) :: CRN a c -> CRN b c
      -> CRN (Either a b) c
left :: CRN a b
     -> CRN (Either a c)
            (Either b c)
right :: CRN a b
      -> CRN (Either c a)
             (Either c b)
entangle :: CRN (Bool, (a,b))
                (Either a b)
```

Figure 3 Functional reactive molecular programming core combinators.

4 Functional Reactive Molecular Programming

The following equivalences form the heart of the functional reactive molecular programming (FRMP) style epitomized by Reactamole:

- Signals, time-varying values, are *interpretations* of collections of species' concentrations as values.
- Signal functions are *typed chemical reaction networks*, a CRN equipped with extra information identifying the species that serve as the inputs and outputs to the function.
- Higher-order signal functions, i.e., signal functions that take other signal functions as input, are *CRN transformers* which produce new CRNs from old ones.

Furthermore, we take inspiration from the arrowized FRP approach of Hughes and thus specify the constructs of FRMP as a collection of *combinators*, higher-order signal functions that users combine to build more complex CRNs from smaller ones.

The heart of FRMP is a compilation pass that transforms these FRP combinators into a CRN that realizes the computation in (abstract) chemistry. We augment the output CRN with *species tags*, additional static information necessary for interpreting the relevant species of the CRN as inputs and outputs to the computation. We call the combination of a CRN with its species tags a *typed chemical reaction network*, formally, a tuple of a CRN N and species tags describing how to interpret its inputs and outputs, written $(N, p^{\mathsf{in}}, p^{\mathsf{out}})$. We represent the compilation process as an interpretation function over signal functions $[\![f]\!] = (N, p^{\mathsf{in}}, p^{\mathsf{out}})$ which denotes that signal function f compiles to typed CRN $(N, p^{\mathsf{in}}, p^{\mathsf{out}})$.

In the following section, we describe this translation for our core combinators. We organize the various combinators by the types that they operate over, recalling that the type `CRN a b` represents a CRN (dually, a signal function) that takes a signal of type `a` as input and produces a signal of type `b` as output. Figure 3 gives an overview of these combinators by category.[2]

[2] We approximate real values using finite-sized Haskell `Double` values.

Species Tags and Signals

We provide one tag for each possible type that we can represent in our implementation. The tags identify the type of signal that the CRN outputs as well as the relevant species that encode that signal.

- The *unit signal tag*, $()$, represents a signal that generates the *unit value*. This is an arbitrary, unique value of that type (written as `()` in Haskell) that acts effectively as a constant that carries no information. Because of this, a unit signal will not be acted on by any reactions and thus does not need an explicit runtime representation in a CRN.
- A *Boolean signal tag*, $(X, \overline{X})$, denotes a Boolean value represented by a pair of species X and $\overline{X}$ in a *dual-rail construction* based on Ellis et al.'s method [7]. These species exhibit an inverse relationship maintained by the constructed CRN as an invariant – when one species has a high concentration, the other has a low concentration. The two species in this dual-rail construction represent `True` and `False`, respectively.
- A *real number tag*, (X^+, X^-), denotes a real value represented by a pair of molecules X^+ and X^-, where we take the value of the real to be the *difference* between the concentrations of X^+ and X^-.
- A *pair signal tag*, (p_1, p_2), denotes the "gluing" together of two signals that operate independently of each other. The components of the tag pair are the tags of the individual signals.
- An *either signal tag*, $(X, \overline{X}, p_1, p_2)$, represents a time-varying discriminated union used in FRMP to achieve dynamic switching. Such a signal is made up of a Boolean signal indicated by species X and $\overline{X}$ as well as two component signals with tags p_1 and p_2. The Boolean species indicates which of the two signals is currently active.

4.1 Algebraic Structures

The core of FRP is composing signal functions together. Suppose that we compose together two signal functions, written `f1 >>> f2`. In FRMP, this amounts to composing the two I/O CRNs representing `f1` and `f2`, call them N_1 and N_2, respectively, by feeding the outputs of N_1 as inputs to N_2. Because the input species of I/O CRNs are catalytic, i.e., the net rate of the input species to N_1 and N_2 is zero, we achieve composition by simply taking the union of the species and reactions of each CRN ($\sqcup$) and then substituting all the input species of f_2 with all the output species of f_1 in the resulting CRN, written $[p_1^{\mathsf{out}} \mapsto p_2^{\mathsf{in}}]$.

$$[\![f_1 \ggg f_2]\!] = ([p_1^{\mathsf{out}} \mapsto p_2^{\mathsf{in}}](N_1 \sqcup N_2), p_1^{\mathsf{in}}, p_2^{\mathsf{out}}) \qquad \text{where} \quad \begin{aligned} [\![f_1]\!] &= (N_1, p_1^{\mathsf{in}}, p_1^{\mathsf{out}}) \\ [\![f_2]\!] &= (N_2, p_1^{\mathsf{in}}, p_2^{\mathsf{out}}). \end{aligned}$$

This is safe because N_2 has no observable effect on the outputs of N_1.

The only caveat we must consider is that the species of the two CRNs are disjoint. Otherwise, unioning their reactions might cause unintended side-effects to their rates. We guarantee this by ensuring that species names are disjoint between CRNs via renaming whenever combining CRNs in this fashion. This is analogous to the notion of α-equivalence in programming languages, where programs are considered equivalent up to renaming of their bound variables.

As a simple example of composition, consider $f_1 = (N_1, (X^+, X^-), (Y^+, Y^-))$ and $f_2 = (N_2, (A^+, A^-), (B^+, B^-))$ where N_1 is the network:

$$X^+ \xrightarrow{1} X^+ + Y^+ \qquad X^- \xrightarrow{1} X^- + Y^- \qquad Y^+ + Y^- \xrightarrow{1} \emptyset,$$

and N_2 is the network:

$$A^+ \xrightarrow{1} A^+ + B^+ \qquad A^- \xrightarrow{1} A^- + B^- \qquad B^+ + B^- \xrightarrow{1} \emptyset.$$

N_1 and N_2 each independently perform an integration of their arguments. The composition of these two CRNs f_1 `>>>` $f_2 = (N_3, (X^+, X^-), (B^+, B^-))$ where N_3 is the network:

$$\begin{array}{lll} X^+ \xrightarrow{1} X^+ + Y^+ & X^- \xrightarrow{1} X^- + Y^- & Y^+ + Y^- \xrightarrow{1} \emptyset \\ Y^+ \xrightarrow{1} Y^+ + B^+ & Y^- \xrightarrow{1} Y^- + B^- & B^+ + B^- \xrightarrow{1} \emptyset. \end{array}$$

N_3 is precisely the union of the reactions of N_1 and N_2 but with the input species of N_2, A^+ and A^- replaced by the output species of N_1, Y^+ and Y^-. Observe that, by virtue of composing two integration functions together, N_3 performs two integrations on its input.

Aggregate data types are represented in Reactamole through *product types*, i.e., tuples. The elements of tuples are, by definition, independent values. Because we maintain disjointness of species names between CRNs, we form tuples by taking the union of these CRNs directly. The split operator between two CRNs, written `f1 *** f2`, unions the species and reactions of the two input CRNs together and creates a new output species tag that identifies the output of the CRN as a tuple.

$$[\![f_1 \texttt{ *** } f_2]\!] = (N_1 \sqcup N_2, (p_1^{\mathsf{in}}, p_2^{\mathsf{in}}), (p_1^{\mathsf{out}}, p_2^{\mathsf{out}})) \qquad \text{where} \begin{array}{l} [\![f_1]\!] = (N_1, p_1^{\mathsf{in}}, p_1^{\mathsf{out}}) \\ [\![f_2]\!] = (N_2, p_2^{\mathsf{in}}, p_2^{\mathsf{out}}). \end{array}$$

For our example, $[\![f_1 \texttt{ *** } f_2]\!] = (N_4, ((X^+, X^-), (A^+, A^-)), ((Y^+, Y^-), (B^+, B^-)))$ where N_4 is simply the union of the unmodified reactions from N_1 and N_2.

In contrast, the *fanout* operator between two CRNs, written `f1 &&& f2`, transforms two CRNs that expect the same input type into a single CRN that sends a single input to the two CRNs. The operator has type `(&&&) :: CRN a b -> CRN a c -> CRN a (b, c)`. We can implement the fanout operator in terms of composition, split, and a `dup` combinator that produces a signal pair where each component is simply the input species.

$$[\![\texttt{dup } f]\!] = (N, p^{\mathsf{in}}, (p^{\mathsf{out}}, p^{\mathsf{out}})) \qquad \text{where } [\![f]\!] = (N, p^{\mathsf{in}}, p^{\mathsf{out}}).$$

We can then define fanout directly as f `&&&` $g =$ `dup` `>>>` $(f$ `***` $g)$. Again, this is all safe because CRNs are catalytic in their inputs. Thus, the duplication of the input signal will not lead to interfering behavior between f and g.

Finally, we can also create CRNs that involve *feedback*, i.e., the CRN depends on its own output, with the `loop` combinator. To understand how `loop` is implemented, it is useful to first analyze its type:

```
loop :: CRN (a, c) (b, c) -> CRN a b
```

`loop` takes in a CRN that expects a pair signal and produces a pair signal. The result is a new CRN where the second component of the input pair is "patched" by the second component of the output pair, both of type `c`. This leaves behind a CRN that expects an `a` as input and produces a `b` as output.

$$[\![\texttt{loop } f]\!] = ([p_c^{\mathsf{out}} \mapsto p_c^{\mathsf{in}}]N, p_a^{\mathsf{in}}, p_b^{\mathsf{out}}) \qquad \text{where } [\![f]\!] = (N, (p_a^{\mathsf{in}}, p_c^{\mathsf{in}}), (p_b^{\mathsf{out}}, p_c^{\mathsf{out}}))$$

4.2 Booleans

Booleans in FRMP are represented by a dual-rail encoding, where one species indicates the `True` value and the other `False`. These species exhibit a strictly inverse relationship (i.e., when one has a low concentration the other has a high concentration) and the value of the Boolean is given by the high species. This dual-rail construction is based on Ellis et al.'s method [7] and is necessary for the result to be measurable by reporter molecules in a biological system, since it is difficult to detect the absence of a species.

Boolean Negation

FRMP supports negation through a reinterpretation of this Boolean encoding. Negation is achieved by swapping the Boolean value that each dual-rail chemical species represents. This allows for efficient negation in a manner that does not require creating an additional CRN.

$$[\![\texttt{not b}]\!] = (N, p^{\mathsf{in}}, (\overline{X}, X)) \qquad \text{where } [\![\texttt{b}]\!] = (N, p^{\mathsf{in}}, (X, \overline{X}))$$

NAND and Other Logic Gates

The foundation for FRMP's support of Boolean circuits and functions is the robust NAND gate introduced by Ellis et al. [7], as shown in Figure 2. This gate is included as the signal function `nand :: CRN (Bool, Bool) Bool`. All other basic logic gates are implemented using `nand` along with `neg` as described in Section 3.

Lifting

We can also lift (i.e., translate) arbitrary pure Boolean functions into CRNs, which allows us to write Boolean functions in a more natural style, e.g., with variables and conditionals. This is possible because any Boolean function can be represented by a finite number of gates. For example, consider the following Haskell function which determines if a decision between three parties is unanimous:

```
unanimous :: Bool -> Bool -> Bool -> Bool
unanimous x y z = if x then (y && z) else not (y || z)
```

We can lift this function into a CRN using the three-input lifting function:[3]

```
arr3Bl :: (Bool -> Bool -> Bool -> Bool) -> CRN (Bool, Bool, Bool) Bool
```

`arr3Bl unanimous` produces a CRN that computes `unanimous`. This CRN is constructed using a series of signal function gates that mirror the given Haskell function's sum of products. The sum of products is found by first constructing a matrix of all possible combinations of inputs to the Haskell function and the resulting outputs. All instances with output value `False` are then filtered out, and the signal function is then constructed using the corresponding series of AND, OR, and NOT gates that represent the sum of products given by the matrix. This lifting process is a useful mechanism to quickly create a complex CRN directly from Haskell code.

[3] The name `arr` comes from the Haskell `Arrow` typeclass that defines `arr` as its own lifting function.

4.3 Real Numbers

We have already demonstrated our ability to compute over real numbers with CRNs in Section 3. For example, the `integrate` and `sin` signal functions correspond to straightforward CRN implementations. The `neg` combinator is a simple reinterpretation of the input CRN's species tag because $X - Y = -(Y - X)$:

$$[\![\texttt{neg f}]\!] = (N, p^{\mathsf{in}}, (X^-, X^+)) \qquad \text{where } [\![\texttt{f}]\!] = (N, p^{\mathsf{in}}, (X^+, X^-))$$

What about other operations, for example, adding together two real signals? It turns out that we cannot craft a CRN that captures the addition of two abstract real signals. But, in our deterministic context, we know that a CRN can be interpreted as an ODE describing the rates of change of each of the species in the system. Under this interpretation, addition of real signals is precisely addition of their respective ODEs.

However, this implementation strategy is fundamentally different than the other CRN operations we have discussed so far. Unlike `integrate`, which treats its input signal as a black box, performing addition and multiplication requires knowing the input signal's ODEs in order to be exact. To prevent unnecessary dependencies, the `add :: CRN (Double, Double) Double` and `mult :: CRN (Double, Double) Double` signal functions in REACTAMOLE are implemented *lazily.* This means that the creation of new species and reactions is delayed until absolutely necessary. In fact, when adding or multiplying real-valued signals, the intermediate sum or product species may be optimized away entirely. For example, in Section 5, we construct a CRN using one application of `add`, two applications of `mult`, and one application of `integrate`, which only generates one pair of species and five reactions. (See Figure 4 for more details.)

Currently, our approach to FRMP does not support lifting real-valued functions to CRNs. However, a large class of functions are known to be produced by analog computers [3], and new techniques are emerging for lifting various real-valued functions to CRNs [12]. We hope to incorporate these ideas into FRMP in the future.

4.4 Switching

An important capability of a functional reactive program is changing the topology of its components at runtime, i.e., *dynamically switching* between different signals. A simple way to encode the behavior is through a conditional, if t then s_1 else s_2, where s_1 and s_2 are arbitrary signals and t is a Boolean signal. As t transitions between truth values, the overall *conditional* signal transitions between s_1 and s_2. From Subsection 4.1, we know that we can carry t, s_1, and s_2 as a triple of signals. We give this aggregate signal the type `Either a b`, where `a` and `b` are the types of s_1 and s_2, respectively. This choice of type comes from the `Either` type in Haskell which encodes a *discriminated union* – a pair of potential values with a *tag* that says which of the two values are present.

However, because the reactions of a CRN are fixed at compilation time, we don't have a built-in mechanism by which a consuming signal can "change" its reactions to go from consuming s_1 to consuming s_2 during runtime. To solve this problem, we then *entangle* the components of the `Either` signal, t, s_1, and s_2, to produce a single signal with our desired conditional semantics. The `entangle :: CRN (Bool, (a,b)) (Either a b)` signal function creates these entanglements and the *fanin* operator creates a signal function that merges two entangled values into one. The fanin operator has the following type signature:

```
(|||) :: CRN a c -> CRN b c -> CRN (Either a b) c
```

`f1 ||| f2` takes two signal functions that take arbitrary types `a` and `b` as input and produce a common output type `c`. Fanin will join the signal functions f_1 and f_2 into an `Either` signal and then merge their outputs to produce a unified signal of type `c`. The Boolean component of the `Either` signal then controls whether the output signal is generated from f_1 or f_2.

This is necessarily a type-directed process; how we combine s_1 and s_2 depends on their encodings and thus their types. For example, consider the case where s_1 and s_2 are both Boolean signals. We can now combine our `Either` signal by observing that:

$$\text{if } t \text{ then } s_1 \text{ else } s_2 \equiv (t \wedge s_1) \vee (\bar{t} \wedge s_2).$$

We can also entangle `Either` signals when s_1 and s_2 are real signals. To do so, we observe a similar equation for reals, recalling that the Boolean signal t is really a pair of species, b and $\bar{b}$, interpreted in a dual-rail style:

$$\text{if } (b, \bar{b}) \text{ then } s_1 \text{ else } s_2 \approx b \cdot s_1 + \bar{b} \cdot s_2.$$

Note that because b and $\bar{b}$ only approximate 0 and 1, the resulting signal is an approximation of the appropriate output of either s_1 or s_2. Such an approximation is necessary because a CRN cannot have a discontinuity in its solution. Finally, with primitives defined, we can entangle aggregate signals such as pairs by simply entangling their components and then pairing together the resulting signals.

Below is a definition of a *rectify*, a signal function that passes through only the positive component of a signal.

```
rectify :: CRN Double Double
rectify = isPos &&& dup >>> entangle >>> (id ||| constRl 0)
```

Here, the sub-expression `id ||| constRl 0` defines a signal function that is either the identity function or the constant zero. Thus, `rectify` is a signal function that behaves like the identity function if the input is positive and otherwise behaves like the constant zero.

The Reactamole Implementation

Reactamole is an implementation of this approach to functional reactive molecular programming as an *embedded* domain-specific language (eDSL) within Haskell.[4] By embedding Reactamole in Haskell, we can take direct advantage of Haskell's existing facilities – language constructs, the type system, and its tooling support – in developing CRNs. To maximize the benefits of this embedding, we choose a different, yet equivalent, adaption of FRP into CRNs. We define signal functions to be *genuine* Haskell functions between signal values, i.e., `data SF a b = SF (Signal a -> Signal b)`, where the `Signal` type corresponds to a CRN. This change in encoding allows us to use Haskell's rich higher-order programming facilities to be able to express combinators more concisely. For example, consider the composition of two CRNs with the composition operator `f1 >>> f2`. In Reactamole, because `f1` and `f2` are now Haskell functions, the composition operator between CRNs is simply the composition operator between Haskell functions, `(.)`.

```
(>>>) :: SF a b -> SF b c -> SF a c
(SF f) >>> (SF g) = SF (g . f)     -- N.B., (.) composes right-to-left
```

Other combinators enjoy simpler implementations than what was presented in this section due to the embedded nature of Reactamole.

[4] Reactamole is available at `https://github.com/digMP/haskell-reactamole`.

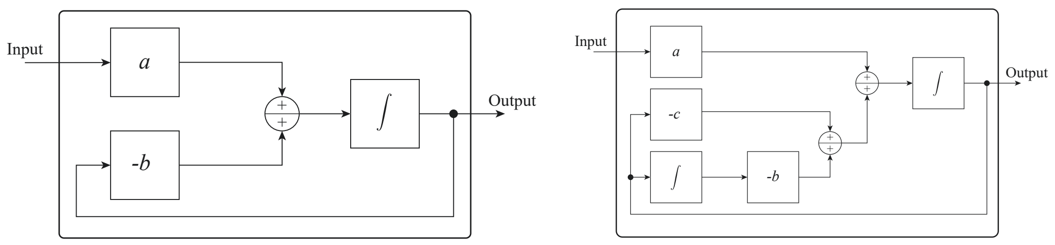

Figure 4 Visualization of the low-pass (left) and band-pass (right) filters in REACTAMOLE. The boxes containing constants a, b, and c correspond to constant multiplier signal functions.

5 Case Study: Amplitude Modulation

We now demonstrate REACTAMOLE's expressiveness via a case study: implementing chemical reaction networks to perform amplitude modulation [19]. Amplitude modulation (AM) is a common technique for sending multiple signals through a shared medium. Intuitively, an AM modulator combines a signal $u(t)$ with a sinusoidal *carrier signal* $s(t)$ via multiplication. Many signals $u_1(t), u_2(t), \ldots$ can then be simultaneously transmitted through a shared medium $m(t)$ by superimposing the modulated signals. In CRNs, modulated signals need only be added into a single signal, represented by a pair of species (M^+, M^-). This is analogous to radio stations transmitting modulated signals at specified frequencies, which all combine in the atmosphere.

We previously saw how to specify a sine wave in REACTAMOLE. It is not difficult to extend this example in order to generate a carrier signal with a given frequency. We first define a helper signal function called `constMult` that multiplies a signal by a constant.

```
constMult :: Double -> CRN Double Double
constMult d = (constRl d &&& id) >>> mult
```

Here, `constRl d` produces a signal function `CRN a Double` that ignores its input and emits a signal with the constant `d`. Using `constMult`, we can now specify a CRN that generates a sine wave with a given frequency.

```
carrier :: Double -> CRN a Double
carrier w = loop (proj2 >>> constMult (-w) >>> integrate 1
                        >>> constMult w    >>> integrate 0
                        >>> dup)
```

Note that this implementation is nearly identical to that of `sin`, defined in Section 3. However, by adding the constant multipliers, `carrier 5` will produce a signal $s(t) = \sin(5t)$. The constant `w` is incorporated into the rate constants of the reactions, so `carrier w` also consists of four species and six reactions.

One common component used to implement AM modulation and demodulation is the *low-pass filter*. A simple first-order low-pass filter is realized by integrating the sum of the input and output multiplied by specific parameters a and b, as shown in Figure 4. By choosing the appropriate parameters, we can generate a low-pass filter with a specific cut-off frequency. For example, if we choose a to be 0.0001, then the cut-off frequency will be 0.01 radians per second. The low-pass filter presented in [19] can be specified as follows:

```
lowPass :: Double -> Double -> CRN Double Double
lowPass a b = loop (constMult a *** constMult (-b) >>> add >>> integrate 0 >>> dup)
```

The I/O CRN generated by `lowPass a b` consists of the following five reactions:

$$X^+ \xrightarrow{a} X^+ + Y^+ \qquad Y^+ \xrightarrow{b} Y^+ + Y^-$$
$$X^- \xrightarrow{a} X^- + Y^- \qquad Y^- \xrightarrow{b} Y^- + Y^+ \qquad Y^+ + Y^- \xrightarrow{1} \emptyset$$

where (X^+, X^-) comprise the input signal. The CRN satisfies the ODE $\frac{dy}{dt} = ax - by$ where $y(t) = y^+(t) - y^-(t)$ and $x(t) = x^+(t) - x^-(t)$.

Another common AM component is the *band-pass filter*, which can be used to select a specific carrier frequency from the medium species and attenuate all other carrier signals present in the medium species. Below is the implementation of the band-pass filter presented in [19], which is also included visually in Figure 4.

```
bandPass :: Double -> Double -> Double -> CRN Double Double
bandPass a b c = loop (first (constMult a)
  >>> second (constMult (-c) &&& (integrate 0 >>> constMult (-b)) >>> add)
  >>> add >>> integrate 0 >>> dup)
```

The I/O CRN generated by `bandPass a b c` consists of the reactions

$$X^+ \xrightarrow{a} X^+ + Y^+ \qquad Z^+ \xrightarrow{b} Z^+ + Y^-$$
$$X^- \xrightarrow{a} X^- + Y^- \qquad Z^- \xrightarrow{b} Z^- + Y^+$$
$$Y^+ \xrightarrow{1} Y^+ + Z^+ \qquad Y^+ \xrightarrow{c} Y^+ + Y^- \qquad Z^+ + Z^- \xrightarrow{1} \emptyset$$
$$Y^- \xrightarrow{1} Y^- + Z^- \qquad Y^- \xrightarrow{c} Y^- + Y^+ \qquad Y^+ + Y^- \xrightarrow{1} \emptyset$$

and satisfies the ODEs $\frac{dy}{dt} = ax - bz - cy$ and $\frac{dz}{dt} = y$. Note that the species (Z^+, Z^-) are internal species to the I/O CRN that are not communicated in its output.

We now show how to modulate and demodulate signals using a carrier frequency. We use the technique in [19] that has the medium species M^+ and M^- and can be specified in Reactamole with:

```
modulate :: Double -> CRN Double Double
modulate w = loop (first (id &&& carrier w >>> mult)
  >>> second neg >>> add >>> integrate 0 >>> dup)
```

Here, `modulate f` produces a pair of species representing the medium $m(t)$ that satisfies $\frac{dm}{dt} = u(t) \cdot \sin(ft) - m(t)$ where $u(t)$ is the input signal. Klinge and Lathrop [19] also proposed a method to superimpose multiple modulated signals through a single medium $m(t)$, which can be easily accomplished in Reactamole with `add :: CRN Double Double`.

Given a signal $m(t)$ that may carry several modulated signals, we now use the same methods in [19] to retrieve a signal. A simple AM demodulator is realized with a band-pass filter to select the desired carrier frequency followed by a function to pass only the positive parts of the signal. A low-pass filter may then be used to remove the carrier frequency to recover an approximation to the original signal.

We can use the `bandPass` and `lowPass` filters, along with `rectify` defined in Subsection 4.4, to extract a signal from a desired carrier frequency.

```
demodulate :: Double -> Double -> CRN Double Double
demodulate w q = bandPass (w/q) (w/q) (w*w) >>> rectify >>> lowPass w w
```

Here, `demodulate w q` generates a CRN that demodulates a signal that has been modulated on a carrier signal at frequency w. The parameter q is used to determine the bandwidth of the band-pass filter, which determines how close two different carriers may be in frequency. Also note that low-pass and band-pass filters may be cascaded to create higher order filters by composing them with the `>>>` combinator.

6 Conclusion

Reactamole unveils new possibilities for the future of molecular programming through exploration of a novel paradigm for the field: functional reactive programming. The language uses typed CRNs – CRNs with extra information about how their chemical species map to Haskell types – to enable complex computation. In particular, Reactamole introduces combinators that allow for computation over basic primitives types as well as the safe manipulation and composition of CRNs.

The representation of CRNs as signal functions in FRP allows for efficient construction of a variety of CRNs using a minimal numbers of species. For example, the NOT gate for Booleans and negation for real values are both achieved through "rewiring" of a signal function (reinterpreting the species' types) and do not require any additional chemical species. This results in OR and AND gates that use the same number of species as the NAND gate they are built from. Additionally, the use of ODEs to represent CRNs enables addition for some CRNs. We also provide a formal construction of the representations of various Haskell types as CRNs, including Booleans, Reals, Pairs, and Eithers. All of these features make Reactamole a useful tool for safe, robust and automated construction and composition of CRNs for a variety of uses.

Future Work

There are four main areas for further improvement that we hope to pursue for Reactamole. First, there is potential for expanding Reactamole's lifting support, including optimizing the construction of lifted Boolean functions, enabling lift for multi-output Boolean functions, and adding support for lifting certain real functions using `Either`. Lifting for reals will require some constraints, as it is not possible to lift real functions whose input signals are not known.

We would also like to improve the handling of approximations and delays. Currently, `nand` and `Either` are approximations because there is some unavoidable delay when working with chemical reactions. In the future, it would be helpful to build out support for tracking approximation margins and specifying additional guarantees. Similarly, composing NAND gates propagates delay and this is currently not controllable by the user because the rate constants are hard-coded. Ellis et al. specify a method for converting a τ value to a rate constant [7] which could be leveraged to allow the user to specify a rate constant for the NAND gates in Reactamole.

It would also be useful to expand the back-end options for Reactamole. Currently, the language interacts directly with MATLAB to simulate CRNs, but this could be expanded by allowing options to export to other tools, such as SimBiology, or through building an embedded ODE solver.

Finally, Reactamole may be a good candidate for a standalone language. In particular, we can move beyond the limitations of Haskell and specialize the language features to the molecular programming setting. With this change, we can consider adding a strong, linear temporal logic-based type system to Reactamole to capture fine-grained correctness properties of our CRNs [18]. Such a type system can, in turn, enable the efficient automatic generation of CRNs from specification, i.e., program synthesis [11].

References

1 Stefan Badelt, Seung Woo Shin, Robert F. Johnson, Qing Dong, Chris Thachuk, and Erik Winfree. A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 232–248, Cham, 2017. Springer International Publishing.

2 Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Comput. Surv.*, 45(4), 2013. doi:10.1145/2501654.2501666.

3 Olivier Bournez, Daniel Graça, and Amaury Pouly. On the functions generated by the general purpose analog computer. *Information and Computation*, 257:34–57, 2017. doi:10.1016/j.ic.2017.09.015.

4 Luca Cardelli. Kaemika App: Integrating Protocols and Chemical Simulation. In Alessandro Abate, Tatjana Petrov, and Verena Wolf, editors, *Computational Methods in Systems Biology*, pages 373–379, Cham, 2020. Springer International Publishing.

5 Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of Chemical Reaction Networks. In Anne Condon, David Harel, Joost N. Kok, Arto Salomaa, and Erik Winfree, editors, *Algorithmic Bioprocesses*, pages 543–584. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88869-7_27.

6 Conal M. Elliott. Push-Pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596638.1596643.

7 Samuel J. Ellis, Titus H. Klinge, and James I. Lathrop. Robust chemical circuits. *Biosystems*, 186:103983, 2019. doi:10.1016/j.biosystems.2019.103983.

8 Irving Robert Epstein and John Anthony Pojman. *An Introduction to Nonlinear Chemical Dynamics: Oscillations, Waves, Patterns, and Chaos*. Oxford University Press, 1998.

9 François Fages, Guillaume Le Guludec, Olivier Bournez, and Amaury Pouly. Strong turing completeness of continuous chemical reaction networks and compilation of mixed analog-digital programs. In Jérôme Feret and Heinz Koeppl, editors, *Computational Methods in Systems Biology*, pages 108–127, Cham, 2017. Springer International Publishing.

10 Martin Feinberg. *Foundations of chemical reaction network theory*. Springer, 2019.

11 Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Synthesizing Functional Reactive Programs. *arXiv:1905.09825 [cs]*, 2019. arXiv:1905.09825.

12 Mathieu Hemery, François Fages, and Sylvain Soliman. Compiling Elementary Mathematical Functions into Finite Chemical Reaction Networks via a Polynomialization Algorithm for ODEs. working paper or preprint, 2021. URL: https://hal.inria.fr/hal-03220725.

13 Xiang Huang, Titus H. Klinge, and James I. Lathrop. Real-time equivalence of chemical reaction networks and analog computers. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 37–53, Cham, 2019. Springer International Publishing.

14 P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, 2002.

15 Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196–es, 1996. doi:10.1145/242224.242477.

16 John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111, 2000. doi:10.1016/S0167-6423(99)00023-4.

17 John Hughes and John O'Donnell. Expressing and reasoning about non-deterministic functional programs. In Kei Davis and John Hughes, editors, *Functional Programming*, pages 308–328, London, 1990. Springer London.

18 Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification - PLPV '12*, page 49, Philadelphia, Pennsylvania, USA, 2012. ACM Press. doi:10.1145/2103776.2103783.

19 Titus H. Klinge and James I. Lathrop. Modulated signals in chemical reaction networks. *CoRR*, abs/2009.06703, 2020. arXiv:2009.06703.

20 Titus H. Klinge, James I. Lathrop, and Jack H. Lutz. Robust Biomolecular Finite Automata. *CoRR*, abs/1505.03931, 2015. arXiv:1505.03931.

21 Simon Marlow. Haskell 2010 language report.

22 Marko Vasic, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular Programming Language. *arXiv:1809.07430 [cs]*, 11145, 2018. doi:10.1007/978-3-030-00030-1.

A Haskell

Haskell is a statically-typed, lazy, pure functional programming language [21] used both in industry for its strong correctness guarantees and in academia as a testbed for programming languages research. Reactamole is an embedded domain-specific language [15] within Haskell and as such, writing programs in Reactamole is equivalent to writing Haskell programs. Through this embedding, Reactamole enjoys the benefits of Haskell's strong static type system to ensure the well-formedness of chemical reaction networks. Here we review the salient features of Haskell necessary to understand our presentation of Reactamole.

A.1 Bindings and Type Signatures

Like other languages, a Haskell program is composed of a collection of top-level bindings, usually function declarations. For example, here is a function that computes a simple numeric result:

```haskell
cToF :: Double -> Double
cToF c = c * 9.0/5.0 + 32
```

A binding typically possesses a *type signature* associated with it that describes the static type of the value bound to that name. In this example, the binding `cToF` has a *function type* `Double -> Double`. The type of value that `cToF` takes as input is given to the left of the arrow (`->`) and the type of its output is given to the right of the arrow. Here, `cToF` is a function that takes a Double as input and produces a Double as output. If a function takes multiple inputs, we separate each input type by an arrow. For example, the less-than comparison operator has the (simplified) type `(<) :: Int -> Int -> Bool`. In other words, less than is a function that takes two arguments, both `Int`s, and produces a `Bool` as output.

The *definition* of the binding follows the type signature. Above, we declare `cToF` as a function, specifying the parameter to the function `c` after the binding's name but before the equals sign. The function body is a single expression (`c * 9.0/5.0 + 32`) whose resulting value is returned when the function is evaluated.

A.2 Polymorphism

Haskell's rich type system allows the programmer to write down functions that take values of *any* type, a feature called *parametric polymorphism.* This is commonly implemented as *generic types* in other languages, such as Java or C#. In Haskell, any name that appears in a type signature that *starts with a lowercase letter* is assumed to be a type variable. For example, this binding:

```haskell
id :: a -> a
id x = x
```

implements the identity function that takes a value of some *arbitrary* type `a` and returns a value of that same type. `id` can be passed any value and the type variable `a` is instantiated to the type of that value, e.g., `id 5` instantiates the `a` to be `Int`.

Type variables also commonly appear as *arguments to other types*. The canonical example of such a parameterized type is the list type, written `[a]`, which corresponds to values that are lists whose elements are all of type `a`. More commonly, you will see parameterized types like the standard data type `Maybe a`, which is, conceptually, a box that *potentially* contains a value of type `a`.

A.3 Functions as Values

Finally, an important concept when working in the abstract setting of functional reactive programming is *functions as first-class values*. Functions can be passed as input to other functions and produced as output. An example of such a *higher-order function* is the composition operator `(.)` which operates analogously to mathematical function composition. We can use composition to "glue" together functions, sending the outputs of one as the inputs of another. For example, here is a simple function that defines `isOdd` by composing an even-testing function and Boolean negation:

```
isEven :: Int -> Bool
isEven x = x ‘div’ 2 == 0

isOdd :: Int -> Bool
isOdd x = (not . isEven) x
```

Note that the output of `(.)` here will be a function that takes an `Int`, feeds it through `isEven`, then takes that resulting `Bool` and feeds it through `not`. However, because this result is a function, it is superfluous to write the argument `x` in the definition of `isOdd` because all we do is pass it to the result of the composition. Instead, we can define `isOdd` *directly* in terms of the composition:

```
isOdd’ :: Int -> Bool
isOdd’ = not . isEven
```

When a function is defined directly without the need to reference its argument, we call such a function *point-free*. Much of the code we write in REACTAMOLE consists of manipulating polymorphic function-like objects in this higher-order, point-free style.

A.4 Example: Interpreting Functional Reactive Code

As an example of applying these concepts toward understanding REACTAMOLE code, let us revisit the `sin` function from Section 3 and analyze how its type relate to its implementation.

```
sin :: CRN a Double
sin = loop (proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup)
```

In doing so, we will effectively walk through the process of *type checking* the code, ensuring that the implementation of `sin` is consistent with its stated type.

Immediately, we can see that `sin` is defined in terms of a call to the `loop` function which has type `loop :: CRN (a, c) (b, c) -> CRN a b`. Without understanding the particulars of `loop`, we can see that `loop` takes a CRN as input and produces a CRN as output. This is good because `sin` is suppose to be a CRN! Furthermore, we can observe that the type of `sin` is `CRN a Double` and the output type of `loop` is `CRN a b`. This means that the type variable `b` will be instantiated to `Double`. Consequently, this means that the output of `loop` has type `CRN a Double`, which is consistent with the declared type of `sin`. Now, we must check that the value passed to loop has type `CRN (a, c) (Double, c)`, i.e., a CRN that takes a pair of an `a` and a `c` as input and produces a pair of a `Double` and a `c` as output.

Next, let's look at the argument to `loop` and work through its type. The composition operator `(>>>)` has type:

```
(>>>) :: CRN a b -> CRN b c -> CRN a c
```

Intuitively, we can think of `(>>>)` as chaining together the output of one CRN of type `b` with a CRN expecting that same type as input. The result is a CRN that takes as input the intended input of the first CRN and produces as output the intended output of the second CRN.

Furthermore, `(>>>)` is left-associative, so the argument is really parenthesized as:

```
((((proj2 >>> neg) >>> integrate 1) >>> integrate 0) >>> dup)
```

Because of this, we first look at the type of the CRN produced by `proj2 >>> neg`. `proj2` and `neg` have the following types:

- `proj2 :: CRN (a, c) c`
- `neg :: CRN Double Double`

The composition operator feeds the output of `proj2` as the input of `neg`, and so it must be the case that the type variable `c` is instantiated to `Double`. Therefore, the type of the overall subexpression is:

```
proj2 >>> neg :: CRN (a, Double) Double
```

In other words, so far, we have a CRN that takes a pair of an unknown type `a` and a `Double` as input and produces a `Double` as output.

Next, we compose this CRN with the CRN produced by `integrate` of type

```
integrate :: Double -> CRN Double Double
```

In other words, `integrate` is a function that, when given a `Double`, produces a CRN that takes a `Double` as input and produces a `Double` as output. Thus, the expression `integrate 1` has type `CRN Double Double`. If we compose this CRN with `proj2 >>> neg`, we obtain a CRN with type:

```
proj2 >>> neg >>> integrate 1 :: CRN (a, Double) Double
```

Furthermore, chaining the second `integrate` call preserves this type, yielding:

```
proj2 >>> neg >>> integrate 1 >>> integrate 0 :: CRN (a, Double) Double
```

Now we need to compose this whole expression with the last call in the chain, `dup`. The function `dup` has type `dup :: CRN d (d, d)` for some unknown type `d`. When we compose this with our result, we take the output of our built-up expression, `Double`, and feed it to `dup`. This leads to a final type for the sub-expression of `loop`:

```
proj2 >>> neg >>> integrate 1 >>> integrate 0 >>> dup
  :: CRN (a, Double) (Double, Double)
```

Finally, we need to reconcile this type with `loop`. Above, we determined that `loop` expects a value of type `CRN (a, c) (Double, c)`. Through our derivation, we have concluded that the argument has type `CRN (a, Double) (Double, Double)`. Consequently, we know that the argument type matches the expected type of the function as long as we instantiate `c` to `Double`!