Reactamole: Functional Reactive Molecular Programming

Titus H. Klinge[1†], James I. Lathrop[1*†], Peter-Michael Osera[2†] and Allison Rogers[2†]

[1*]Computer Science Department, Iowa State University, Ames, 50011, IA, USA.
[2]Computer Science Department, Grinnell College, Grinnell, 50112, IA, USA.

*Corresponding author(s). E-mail(s): jil@iastate.edu;
Contributing authors: tklinge@iastate.edu; osera@cs.grinnell.edu; rogersal@grinnell.edu;
†These authors contributed equally to this work.

Abstract

Chemical reaction networks (CRNs) are an important tool for molecular programming. This field is rapidly expanding our ability to deploy computer programs into biological systems for various 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 more straightforward computation with CRNs. Recently, research has been conducted into various 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 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. We also show how Reactamole can be used to specify abstract CRNs whose structure depends on the reactions and species of its input, allowing users to specify more general CRN behaviors.

Keywords: Molecular programming, functional reactive programming, chemical reaction networks, general-purpose analog computing, type systems, functional programming

MSC Classification: 68N15 , 68N18 , 92C99

1 Introduction

Molecular programming harnesses computer science toward 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 networks (CRNs), abstractions of chemical reactions [9, 16, 19]. 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 [3, 17]. CRNs

are also closely related to other models of computing such as population protocols [1, 12] and the general purpose analog computer [5, 37].

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 simultaneously depending on the availability of the reactants. This creates race conditions that can make coding in this framework more difficult and error-prone [40]. CRNs are also unstructured and not easily composed; adding a single reaction can radically change its behavior. These challenges motivate creating 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++ [40] and Kaemika [8]. The Biochemical Abstract Machine (BIOCHAM) [7, 18] also implements several languages to support modeling, analyzing, and verifying CRNs. 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 [40]. Kaemika supports specifying CRNs in a functional style, including support for high-order functions and recursion [8]. Both CRN++ and Kaemika allow side effects, with Kaemika depending on these side effects to generate new species within functions. BIOCHAM provides an array of tools for modeling CRNs, including a language for analyzing, simulating, and verifying biochemical models and another for using temporal logic to formally specify biological processes gleaned from experimentation [7]. BIOCHAM also supports robustness optimizations [18] and synthesis of CRNs from GPAC circuits and polynomial differential e quation systems, among other features. However, BIOCHAM does not focus on providing linguistic tools for designing, composing, and performing computation with CRNs, which is our aim. Additionally, these implementations leave room to improve the synchronicity between language structure and CRN behavior.

We address these challenges 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 [4]. In FRP, systems are modeled as graphs where nodes are operations, and edges indicate how data flows between these operations, focusing 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. Thus, they can 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 [23]. Furthermore, by combining FRP and CRNs, we open the door for applying recent advancements in programming language theory to the development of CRNs. For example, in this FRMP paradigm, we can now consider integrating type systems that verify the safety properties of functional reactive programs [26] or program synthesizers for functional reactive programs [20].

Reactamole can also be used to specify behaviors that cannot be represented by a single CRN. For example, suppose we want a CRN that is given a signal $x(t)$ as input and produces an output signal $y(t)$ that satisfies $y(t) = 2x(t)$ for all time $t \in \mathbb{R}_{\geq 0}$. It is easy to show that no *single* CRN can compute the signal $y(t)$ for all possible inputs $x(t)$ because it requires foreknowledge of the input signal. In fact, simply ensuring that $y(0) = 2x(0)$ requires knowing the initial value of

X prior to initializing Y. Nevertheless, Reactamole is capable of expressing behaviors akin to $y(t) = 2x(t)$ by using a higher-order encoding of the CRN and instantiating its reactions and initial conditions only after it is combined with a CRN that produces its input signal $x(t)$. This makes Reactamole more expressive and efficient, since this translation process can eliminate unnecessary intermediate species and reactions, reducing the overall size of the final CRN program.

We organize the paper as follows. In section 2, we review the basic definitions o f c hemical 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. In section 5, we demonstrate the potential of Reactamole by way of a case study—implementing amplitude modulation over real-valued signals. We showcase the ability of Reactamole to specify abstract families of CRNs in section 6 and how Reactamole optimizes the number of species and reactions of a CRN. Finally, in section 7 and section 8 we review related work and conclude with final remarks.

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.

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 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 [28] 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. A reaction is a triple $\rho = (\mathbf{r}, \mathbf{p}, k)$, where $\mathbf{r} \in \mathbb{N}^S$ is the reactant vector, $\mathbf{p} \in \mathbb{N}^S$ is the product vector, and $k \in (0, \infty)$ is the rate constant. Note that $\mathbb{N}^S$ is the set of functions mapping species to non-negative integers, and we do not allow $\mathbf{r} = \mathbf{p}$. In this paper, we usually represent CRNs simply as a set of reactions that implicitly define a set of species. For example, the reaction

$$X + U \xrightarrow{5} 2X$$

can be interpreted as the CRN $N = (S, R)$ with $S = \{X, U\}$ and $R = \{(\mathbf{r}, \mathbf{p}, k)\}$, where $\mathbf{r}(X) = \mathbf{r}(U) = 1$, $\mathbf{p}(X) = 2$, $\mathbf{p}(U) = 0$, and $k = 5$.

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 its 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)

$$\frac{dx}{dt} = 5xu, \qquad \frac{du}{dt} = -5xu.$$

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 simply x to denote the concentration of species X at the time t.

Similar to Klinge, Lathrop, and Lutz [28], 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 catalysts in any reaction, which prevents the I/O CRN from inducing unwanted side affects on its input signal. This is a property we leverage in Reactamole to implement composition of CRNs. We also use the fact that a CRN $N = (S, R)$ is equivalent to the I/O CRN $N = (S, R, \emptyset, S)$ with no input species.

We also note that deterministic CRNs are closely related to Shannon's *general purpose analog computer* (*GPAC*) [5, 6, 37], which can also be regarded as a system of polynomial ODEs. In fact, deterministic CRNs effectively express a subset of the GPAC, since they are only capable of inducing ODEs of the following form:

$$\frac{dx}{dt} = P - xQ,$$

where P and Q are polynomials with positive coefficients over the species in the CRN. The $x(t)$ factor in the negative component of the ODE is necessary, because CRNs can only destroy molecules if they participate in the reaction as a reactant. However, the GPAC and deterministic CRNs are now known to be *exactly equivalent* models [17] if a GPAC variable $x(t)$ is represented with two species X^+, X^- where at all times t we maintain the invariant $x(t) = x^+(t) - x^-(t)$. This is often called a *dual-rail* encoding of the variable $x(t)$. We leverage this equivalence in Reactamole to optimize the number of reactions and species in CRNs by encoding CRNs as GPAC initial value problems and only translating them into a CRN when necessary using the method of [17].

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 [25].

Functional Reactive Programming (FRP) [14] 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 [24], 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 following.

```
(***) :: SF a b -> SF c d -> SF (a, c) (b, d)
(&&&) :: SF a b -> SF a c -> SF a (b, c)
loop  :: SF (a, c) (b, c) -> SF a b
```

The *split operator*, written `f *** g`, combines two input signal functions `f` and `g` and creates a signal function whose inputs and outputs are *pairs* drawn from `f` and `g`, as shown in Figure 1.

The *fanout operator*, written `f &&& g`, 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, as shown in Figure 1.

The *loop operator*, written `loop f`, creates a *feedback* loop where the second component of the output of the signal function is given to itself as input, as shown in Figure 1. In traditional FRP, the `loop` combinator is a *fixed p oint operator*, allowing recursive signal function definitions. For CRNs, the `loop` combinator enables output species to depend on themselves. Without it, reactions like $X + Y \xrightarrow{1} X$ where Y is an *output species* would not be expressible, since the concentration of Y depends on itself. In particular, the output Y is consumed at a rate proportional to its own concentration.

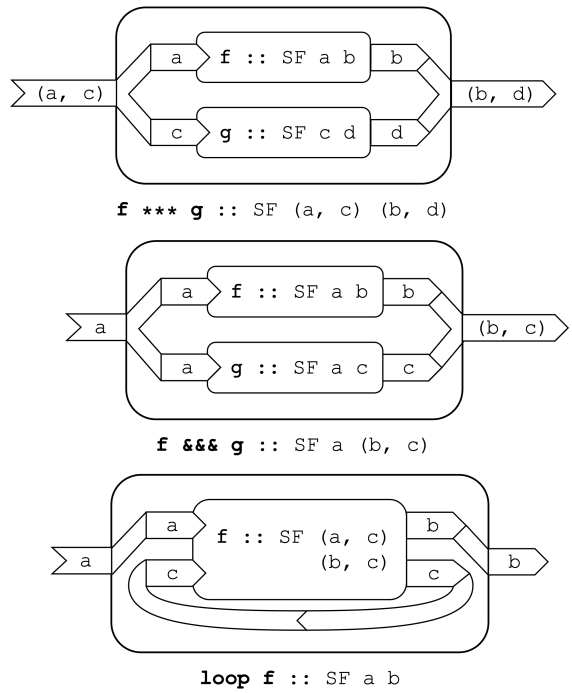

Fig. 1: Arrow combinator visualizations.

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.

Recall that an *input/output chemical reaction network* (*I/O CRN*) is a CRN with some species reserved as "inputs" that can only be used catalytically and some species labeled "outputs." As a result, I/O CRNs are literally signal functions in the functional reactive programming sense, transforming an input signal into an output signal. Although later we will see that Reactamole's signal functions are more abstract than I/O CRNs, it is helpful to regard a signal function as literally an I/O CRN. We now explore the expressive power of arrowized I/O CRNs through several examples.

We begin with I/O CRNs that produce real numbers. Since species concentrations are nonnegative, we encode a real number as the difference between two species using the dual-rail encoding of [17]. 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 a primitive:

```
integrateSF :: Double -> SF Double Double
```

`integrateSF` 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 $y(t)$. Intuitively, `integrateSF y0` can be regarded as an I/O CRN consisting of the three reactions:

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

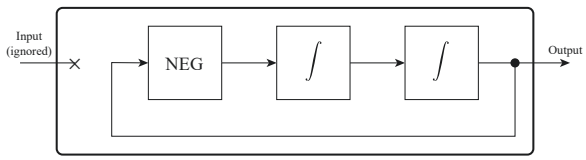

Fig. 2: Visual representation of the arrowized `sin` implementation.

The third reaction has no effect on the encoded value of $y(t)$ but helps bound the concentrations of Y^+ and Y^-. The induced ODEs according to the law of mass action are:

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

Since the species are dual-railed, 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 observe the solution is indeed $y(t) = \int_0^t x(s)ds + y_0$.

Other primitives we provide in Reactamole are shown in Figure 6, but we highlight a few particularly relevant ones here.

- `negateSF` performs numerical negation by reversing the roles of X^+ and X^-. As a result, it does not generate additional reactions as it simply *reinterprets* the output species of the CRN.
- `idSF` is the identity signal function and produces its inputs as outputs, unmodified.
- `proj1SF` and `proj2SF` take *pairs* of signals as inputs and project out the individual components of those pairs as output.
- `dupSF` 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.

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 2:

```
sin :: SF a Double
sin = loop (proj2SF >>> negateSF >>>
  integrateSF 1 >>> integrateSF 0 >>> dupSF)
```

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. We will see later that leaving the input type generic makes `sin` easier to combine with other signal functions rather than explicitly requiring that the input signal be empty.

Since `sin` uses two applications of `integrateSF`, the resulting CRN would consist of six reactions:

$$X_1^+ \xrightarrow{1} X_1^+ + X_2^- \qquad X_2^+ \xrightarrow{1} X_2^+ + X_1^+$$
$$X_1^- \xrightarrow{1} X_1^- + X_2^+ \qquad X_2^- \xrightarrow{1} X_2^- + X_1^-$$
$$X_2^+ + X_2^- \xrightarrow{1} \emptyset \qquad X_1^+ + X_1^- \xrightarrow{1} \emptyset$$

The `loop` combinator creates a feedback loop that allows the signal $x(t)$ to depend on itself. If you regard each wire in Figure 2 as a variable that consists of two species, then the loop combines two wires, reducing the number of species by two by reusing the same species names. Without it, the CRN would have *three* pairs of species instead of two—one for each wire before, in-between, and after the integrators.

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

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

and therefore $x_1(t) = \sin(t)$ and $x_2(t) = \cos(t)$. Note that the initial conditions of x_1 and x_2 are implicitly provided via parameters to `integrateSF`.

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 `notSF :: SF Bool Bool` and

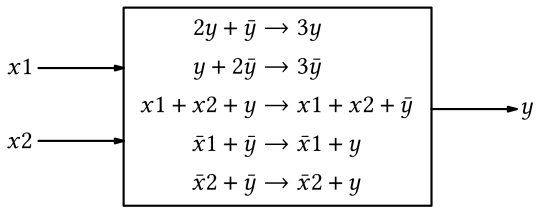

Fig. 3: REACTAMOLE `nandSF` implementation.

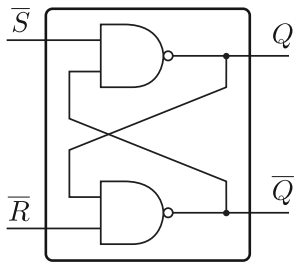

Fig. 4: Traditional `srLatch` circuit.

`nandSF :: SF (Bool, Bool) Bool`. Similar to the `negateSF` signal function for real numbers, `notSF` "crosses the wires" of X and $\overline{X}$ without adding any additional species or reactions. The `nandSF` signal function consists of five reactions, using the implementation provided by Ellis, Klinge, and Lathrop [15] and visualized in Figure 3.

Using these primitives, we can define the other elementary gates, which all have the type `SF (Bool, Bool) Bool`.

```
orSF   = (notSF *** notSF) >>> nandSF
andSF  = nandSF >>> notSF
norSF  = orSF >>> notSF
xorSF  = (nandSF &&& orSF) >>> andSF
xnorSF = xorSF >>> notSF
```

Note that `notSF` does not introduce additional overhead, so the signal functions `orSF`, `andSF` and `norSF` are no more complex than `nandSF`.

We can also use the `loop` combinator to create sequential logic gates such as a *set-reset latch* which is visualized in Figure 4.

```
srLatch :: SF (Bool, Bool) (Bool, Bool)
srLatch = loop (crossWires >>>
  (nandSF *** nandSF) >>> dupSF)
```

Here, `crossWires` is a simple signal function that "rearranges the wires" so that the outputs are looped back into the appropriate NAND gates. This is implemented similarly to `notSF`, making it a zero-overhead signal function that simply reinterprets the species.

Since `srLatch` has two Boolean inputs and is implemented with two `nandSF` gates, the resulting I/O CRN consists of eight species and nine reactions. The final CRN has nine reactions instead of ten because REACTAMOLE generates a GPAC initial value problem before creating the final CRN and combines identical ODE terms into a single reaction. Intuitively, this means that reactions that share the same left-hand-side species (*i.e.*, reactants) will be automatically combined.

Finally, we can create signal functions that employ both real-valued signals and Boolean signals. For example, we include a primitive `isPosSF :: SF Double Bool` that tests if a real-valued input is positive. Note that `isPosSF` is a continuous approximation of a discontinuous function, so its Boolean output is meaningless if the input signal is close to zero. Intuitively, it is implemented in a similar fashion to the NAND gate by creating two new species for the Boolean output and the following four reactions:

$$2Y + \overline{Y} \xrightarrow{1} 3Y$$
$$2\overline{Y} + Y \xrightarrow{1} 3\overline{Y}$$
$$X^+ + \overline{Y} \xrightarrow{1} X^+ + Y$$
$$X^- + Y \xrightarrow{1} X^- + \overline{Y}$$

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

```
clock :: SF a Bool
clock = sin >>> isPosSF
```

The solution of selected species in the I/O CRN specified by `clock` is visualized in Figure 5.

Then `clock` could be combined with `srLatch` to create chemical flip flops, memory units, or multi-stage molecular programs.

4 Functional Reactive Molecular Programming

We now describe the heart of the functional reactive molecular programming (FRMP) paradigm and our implementation of it in REACTAMOLE.

Recall that a *signal* in functional reactive programming is a time-varying value of some type.

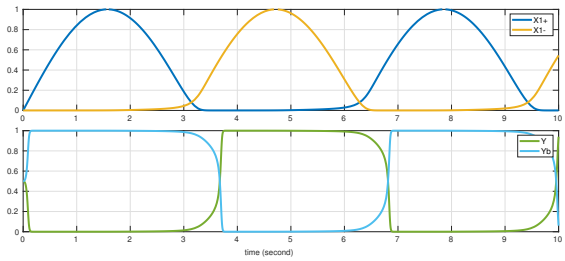

Fig. 5: Plot of selected species of the `clock` signal. The top shows the solutions of X_1^+ and X_1^-, *i.e.*, the real-valued encoding of the $\sin(t)$ function. The bottom shows the solutions of Y and $\overline{Y}$, *i.e.*, the Boolean encoding of the output of the `isPosSF` signal function, given $\sin(t)$ as input.

In FRMP, a `Signal a` encapsulates a collection of time-varying species' concentrations, along with an *interpretation* of those concentrations as a value of type `a`. Intuitively, a `Signal a` is a single chemical reaction network with a fixed initial condition and no external input. According to the law of mass action, such a CRN induces an *initial value problem* (*IVP*) of ordinary differential equations (ODEs) with a unique solution—the real valued signal. Later, we will see that REACTAMOLE implements a `Signal a` directly as a system of ODEs with an initial condition, and employs the equivalence of the GPAC and CRNs to convert it to a CRN when needed using the method of [17].

A *signal function* `SF a b` in FRMP transforms a signal of type `a` into a signal of type `b`. Intuitively, an `SF a b` can be thought of as an abstract input/output chemical reaction network (I/O CRN) that converts an input signal encoding values of type `a` into a signal encoding values of type `b`. For example, the `nandSF :: SF (Bool, Bool) Bool` represents an I/O CRN that receives two Boolean input signals and produces an output signal representing the NAND of its inputs. In general, REACTAMOLE implements an `SF a b` as a function `Signal a -> Signal b`, allowing the existence of signal functions like `notSF`, which changes the interpretation of the species concentrations without introducing new species or reactions.

A *combinator* is a higher-order signal function, *i.e.*, a signal function that takes other signal functions as input. In FRMP, combinators are essentially I/O CRN *transformers* which produce new I/O CRNs from old ones. Furthermore, we take inspiration from the arrowized FRP approach of Hughes, specifying the constructs of FRMP as a collection of combinators to build larger molecular programs from smaller ones.

The heart of FRMP is a compilation pass that transforms these FRP combinators into an I/O CRN that realizes the computation in (abstract) chemistry. We augment the I/O 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 an I/O CRN with its species tags a *typed input/output chemical reaction network*, formally, a tuple of the I/O 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 $(N, p^{\mathsf{in}}, p^{\mathsf{out}})$.

We describe this translation for our core combinators, organizing them by the types that they operate over. Figure 6 gives an overview of these combinators by category. 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, and 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 `Signal ()` can be regarded as the "empty signal," and therefore the signal function `SF () a` can be regarded as an I/O CRN with no input.

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 representation* based on Ellis *et al.*'s method [15]. These species exhibit an inverse relationship maintained 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

```
-- Algebraic structures
(>>>)  :: SF a b -> SF b c -> SF a c
(***)  :: SF a b -> SF c d -> SF (a, c) (b, d)
(&&&)  :: SF a b -> SF a c -> SF a (b, c)
first  :: SF a b -> SF (a, c) (b, c)
second :: SF a b -> SF (c, a) (c, b)

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

-- Booleans
notSF  :: SF Bool Bool
nandSF :: SF (Bool, Bool) Bool
arr1Bl :: (Bool -> Bool) -> SF Bool Bool

-- Reals
integrateSF :: Double -> SF Double Double
negateSF    :: SF Double Double
addSF       :: SF (Double, Double) Double
multSF      :: SF (Double, Double) Double
isPosSF     :: SF Double Bool
```

Fig. 6: Functional reactive molecular programming core combinators. Note that REACTAMOLE approximates real values using finite-sized Haskell `Double` values.

difference between the concentrations of X^+ and X^-, respectively.

A *pair signal tag*, (p_1, p_2), represents two parallel interpretations of the concentrations of a signal. he 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 indicates which of the two interpretations 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 ($\sqcup$). We then substitute the input species of f_2 with all the output species of f_1 in the resulting CRN, written $[p_1^{\mathsf{out}} \mapsto p_2^{\mathsf{in}}]$.

$$[\![f_1 \texttt{ >>> } f_2]\!] = ([p_1^{\mathsf{out}} \mapsto p_2^{\mathsf{in}}](N_1 \sqcup N_2), p_1^{\mathsf{in}}, p_2^{\mathsf{out}})$$

$$\text{where } \begin{matrix}[\![f_1]\!] = (N_1, p_1^{\mathsf{in}}, p_1^{\mathsf{out}}) \\ [\![f_2]\!] = (N_2, p_2^{\mathsf{in}}, p_2^{\mathsf{out}}).\end{matrix}$$

This is provably 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 behavior. 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 consists of the reactions:

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

and N_2 consists of the reactions:

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

Note that N_1 and N_2 each perform integration on their input signals. The composition of these two CRNs f_1 `>>>` $f_2 = (N_3, (X^+, X^-), (B^+, B^-))$ where N_3 consists of the reactions:

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

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 FRMP through *product types*, *i.e.*, tuples. The elements of tuples are, by definition, independent values. Because we maintain the disjointedness 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` takes two I/O CRNs, `f1 :: SF a b` and `f2 :: SF c d` and creates a CRN that takes a pair as input and a pair as output. The components of the input pair are drawn from the inputs of `f1` and `f2`, and the components of the output pair are drawn from the outputs of `f1` and `f2`. We accomplish this behavior, by taking the union of the species and reactions of the two input CRNs and creating a new output species tag that identifies the output of the CRN as a tuple.

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

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

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

We can then define fanout directly as f `&&&` $g =$ `dupSF >>>` $(f$ `***` $g)$. Again, this provably yields the desired behavior because of the catalytic restriction of I/O CRNs on their inputs. Thus, the duplication of the input signal will not lead to interfering behavior between f and g.

In addition to split and fanout, the operator `first f` takes an I/O CRN performing a computation `f` from `a` to `b` (of type `SF a b`) as input and produces an I/O CRN that takes a pair as input (of type `(a, c)`) and produces a pair as output (of type `(b, c)`).

- The first component of the input/output pairs carries the computation from `a` to `b`.
- The second component of the input/output pairs carries a third value of type `c` unmodified in the computation.

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

Analogously, `second f` performs the same transformation but carries the computation of `f` through the second component of the pair, leaving the first unmodified.

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

```
loop :: SF (a, c) (b, c) -> SF 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.

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

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

The `loop` operator is what allows FRMP to perform non-trivial computations by introducing species whose concentration depends on itself. This feedback mechanism has the same behavior as general purpose analog computing (GPAC) flow diagrams such as shown in Figure 2

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 [15] and allows 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.

$$[\![\texttt{notSF b}]\!] = (N, p^{\mathsf{in}}, (\overline{X}, X))$$
$$\text{where } [\![\texttt{b}]\!] = (N, p^{\mathsf{in}}, (X, \overline{X}))$$

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.* [15], as shown in Figure 3. This gate is included as the signal function `nandSF :: SF (Bool, Bool) Bool`. All other basic logic gates are implemented using `nandSF` along with `notSF` as described in section 3.

4.3 Real Numbers

We have already demonstrated our ability to compute over real numbers with CRNs in section 3. For example, the `integrateSF` signal function has a simple CRN representation. Similarly, `negateSF` is a simple reinterpretation of the input CRN's species tag because $X - Y = -(Y - X)$:

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

What about other operations, for example, adding together two real signals? It turns out that we cannot craft a single, concrete CRN that captures the addition of two abstract real signals. To see why, suppose we have two real-valued signals $x(t)$ and $y(t)$, and we wish to produce a new signal $z(t)$ that satisfies $z(t) = x(t) + y(t)$ for all times t. To achieve this, z must satisfy the initial condition $z(0) = x(0) + y(0)$ and conform to the ODE $z'(t) = x'(t) + y'(t)$. Thus, if x and y are not known in advance, then z cannot be produced exactly. However, if their initial conditions and ODEs *are* known, then producing z is as simple as adding their initial conditions and ODEs.

Reactamole supports operations such as addition, but only by virtue of representing a signal function `SF a b` as *literally* a function `Signal a -> Signal b`. This representation allows us to implement the primitive `addSF :: SF (Double, Double) Double` by creating the ODE of the output variable as a function of the ODEs of the two input variables, maintaining that $z'(t) = x'(t) + y'(t)$. As a result, `addSF` can be regarded as an *abstract* I/O CRN, whose reactions are only instantiated when it is combined with another CRN. This is in contrast to primitives such as `nandSF` that are implemented in a way that is agnostic to its input signal and can be regarded as a concrete I/O CRN.

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 is 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 :: SF (Bool, (a,b)) (Either a b)` signal function creates these entanglements via a simple reinterpretation of the species tag and does not induce additional overhead. Thus, a `Signal (Either a b)` is simply a CRN that simultaneously produces three sub-signals—one for the `Bool` switching signal, one for the `a` signal, and one for the `b` signal. Semantically, downstream computations should regard `a` as the output if the `Bool` is true, otherwise consider `b` as the output.

An important switching operator is the *branch* combinator which has type signature

```
(+++) :: SF a b -> SF c d
      -> SF (Either a b) (Either c d)
```

Similar to the `***` combinator, the I/O CRN `f +++ g` simply joins the I/O CRNs `f` and `g` so their computations happen simultaneously. However, the semantic meaning of `f +++ g` differs from `f *** g` in that the inputs and outputs of the former should be regarded as dynamically switching between the behavior of `f` and `g` and the inputs and outputs of the latter should be regarded as `f` and `g` performing parallel computations.

The primary complexity of switching in FRMP comes from the *fanin* operator, which creates a signal function that *merges* two entangled values into one. The fanin operator has the following type signature:

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

Fanin 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 into an `Either` signal and then merge their outputs to produce a unified signal of type `c`. The Boolean component t of the `Either` signal then controls whether the output signal is generated from the first or second function. In fact, the fanin operator is implemented in Reactamole as:

```
f ||| g = (f +++ g) >>> mergeSF
```

Here, `mergeSF :: (Either a a) a` encapsulates merging two entangled values of type `a`—which exist in parallel—into a *single* value of type `a`.

Merging signals is necessarily a type-directed process, since how we combine two signals depends on their encodings and thus their types. For example, consider the case of merging two Boolean signals, letting t, s_1, and s_2 be the conditional, the consequent, and the alternative signals of the `Either Bool Bool`, respectively. Then we can merge these signals into one by observing:

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

Thus, Reactamole implements `mergeSF` on `Bool`s using Boolean primitives, which induces an overhead of three uses of `nandSF` I/O CRNs.

To merge two real signals, Reactamole employs the GPAC switching function designed by Bournez, Graça, and Pouly (Section 5.3 of [6]), which approximates the switch with controllable precision. Approximation is necessary in this case because a CRN cannot have a discontinuity in its solution.

Aggregate signals such as pairs and eithers are merged by recursively merging their individual sub components.

Reactamole also includes the following combinator for entanglement.

```
split :: SF a Bool -> SF a (Either a a)
split f = f &&& (idSF &&& idSF) >>> entangle
```

The `split` combinator lifts a `Bool` signal into an `Either` signal, which makes it simpler to employ choices in Reactamole. Consider the following definition of *rectify*, a signal function that passes through only the positive component of a signal.

```
rectify :: SF Double Double
rectify = split isPosSF >>> (idSF ||| constRl 0)
```

Note that `split isPosSF` takes a `Double` signal as input and produces an entangled `Double` as output, conditioned on whether the input signal is positive or negative. Moreover, the sub-expression `(idSF ||| 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.

4.5 The Reactamole Implementation

We now discuss some implementation details behind Reactamole[1]. We chose to embed Reactamole in Haskell to employ Haskell's existing facilities—language constructs, the type system, and its tooling support—in developing CRNs. To maximize the benefits of this *shallow embedding* of Reactamole in Haskell [21], we designed the core datatypes to be as general as possible.

Our first generalization was to implement a `Signal a` in terms of ordinary differential equations (ODEs) rather than chemical reactions directly:

```
data Signal a = Sg (Tag a) [ODE]
```

Here a `Tag a` encapsulates the static typing information of the signal and the list of `ODE`s explicitly defines a set of variables, their derivatives, and their initial conditions.

The `Tag a` algebraic datatype has numerous constructors, each of which corresponds to one of the species tags described in subsection 4.1. For example, `NullT` constructs a null tag `Tag ()` and `BoolT x x'` takes two variables as arguments that represent the two "rails" of the dual-rail Boolean encoding and constructs a `Tag Bool`.

Signal functions are represented as genuine Haskell functions between signal values with a type signature:

```
data SF a b = SF (Signal a -> Signal b)
```

This 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 I/O CRNs with the composition operator `f1 >>> f2`. In Reactamole, because `f1` and `f2` are now literally Haskell functions, the `>>>` operator 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
```

It also allows developers to implement "abstract" I/O CRNs such as `addSF`, whose output ODE can be constructed from the ODEs of the input signal. We discuss some more advanced uses of these abstract I/O CRNs in section 6.

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

Reactamole also includes some other optimizations to reduce the complexity of the final CRN program, including:

1. Each variable / ODE is initially *single-railed*, and only converted into its dual-rail, non-negative encoding when translating it into a concrete CRN. This makes Reactamole usable as a FRP language for general purpose analog computer (GPAC) programs without constraining the user to a dual-rail encoding. It also reduces the complexity of CRNs by keeping ODEs concise, thereby reducing the number of reactions generated by signal functions like `addSF` that depend on ODEs.
2. The `RealT` constructor for `Tag Double` takes a *polynomial* of variables and is converted into a single-rail or dual-rail representation only when necessary. This allows real-valued operations to combine without generating a new variable every time. As a result, a compound real-valued signal function such as `(addSF *** addSF) >>> multSF` will only generate a single new variable instead of three. This dramatically reduces the number of species and reactions in CRN programs.

Reactamole includes the functions `toIVP` and `toCRN` to export a signal function `SF a b` to an initial value problem or a CRN. For example, `toIVP (integrateSF 0)` produces the single-railed system of ODEs:

```
INPUT:
  Real(+1.0*x0)
OUTPUT:
  Real(+1.0*x1)
ODEs:
  x1(0) = 0.0, dx1/dt = [+1.0*x0]
```

Notice that `x0` is an input signal to the IVP and therefore has no corresponding ODE.

Similarly, `toCRN (integrateSF 0)` produces:

```
INPUT:
  Real(+1.0*x0 -1.0*x1)
OUTPUT:
  Real(+1.0*x2 -1.0*x3)
REACTIONS:
  x0 --{1.0}-> x2 + x0
  x1 --{1.0}-> x3 + x1
  x2 + x3 --{100.0}->
INITIAL CONDITIONS:
  x2(0) = 0.0
  x3(0) = 0.0
```

The CRN is produced from the initial value problem by first converting it into its dual-rail representation and then translating its terms into chemical reactions. As a result, the input signal is now represented by two species, `x0` and `x1`, and their difference is the true value of the input signal.

4.6 Lifting Pure Functions

While the algebraic constructs of Reactamole are sufficiently expressive, it is useful to be able to express molecular computation using more conventional means. In an ideal world, we could write a regular (Haskell) function and have it "just work" when translated to a molecular context. Recall that in our reactive functional perspective on molecular programming, we view molecular computations as time-varying functions, *i.e.*, signal functions. A regular or *pure* function can be thought of as a time-varying function that simply ignores time. In this sense, we expect the translation of pure functions to the molecular context, a process called *lifting*, to be trivial.

This is the case in a general-purpose abstract FRP framework. The `arr` combinator of type `(a -> b) -> SF a b` takes a pure function as input and produces a signal function as output. The name "`arr`" comes from the Haskell `Arrow` typeclass that defines `arr` as its own lifting function. The signal function `arr f` simply behaves like the pure function `f :: a -> b` "for all time."

Since FRMP represents a signal function as a system of equations with a species tag, where the tag indicates how to interpret the equations, it is difficult to treat a pure function as a black box. Instead, we must be aware of the types of the inputs and outputs of the function to create an appropriate typed CRN. For now, Reactamole only supports type-aware lifting on Boolean-to-Boolean functions. However, a large class of real-valued functions are known to be computable via analog computers [6], and new techniques are emerging for lifting various real-valued functions to CRNs [22]. We hope to incorporate these ideas into Reactamole in future work.

Lifting Boolean Functions

Lifting (*i.e.*, translating) arbitrary pure Boolean functions into signal functions is possible because any Boolean function can be represented by a finite number of Reactamole gates, derived from the sum of products form of the function. In Reactamole, the sum of products is found by first constructing a matrix, or truth table, 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. Specifically, false inputs are run through NOT gates, then all inputs in the same row are combined with AND gates. Finally, these row results are ORed together to produce the output Boolean value. This results in a finite number of gates, as the matrix size is finite.

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 = x == y && y == z
```

We can lift this function into a signal function using the three-input lifting function `arr3Bl`:

```
unanimousSF :: SF (Bool, Bool, Bool) Bool
unanimousSF = arr3Bl unanimous
```

This produces a signal function that computes `unanimous`. Since Reactamole is incapable of lifting general functions, we include many variants of the classical `arr` lifting combinator, including `arr2Bl`, `arr3Bl`, and `arr4Bl`, which can lift 2-ary, 3-ary, and 4-ary Boolean functions into an equivalent Reactamole signal function. The corresponding CRN can be displayed with `toCRN unanimousSF`, resulting in the CRN on the left-hand-side of Figure 7. We discuss the `toCRN` function, which displays the underlying reactions of a CRN, towards the end of this section.

For comparison, consider this hand-coded version of `unanimous` as a signal function using other Reactamole primitives:

```
unanimousSF' :: SF (Bool, Bool, Bool) Bool
unanimousSF' = tup3ToPairSF >>>
  (second proj1SF >>> xnorSF
  &&& (proj2SF >>> xnorSF) >>> andSF)
```

Note that `tup3ToPairSF` converts the input three-tuple `SF (Bool, Bool, Bool)` to a nested pairs representation `SF(Bool, (Bool, Bool))` . The behavior of this signal function is less apparent; nevertheless, it has comparable behavior to `unanimousSF` defined above. Moreover, it mirrors the original

```
INPUT:
  Tuple(Bool(x0, x1),
        Bool(x2, x3),
        Bool(x4, x5))
OUTPUT:
  Bool(x14, x15)
REACTIONS:
  x0 + x7 + x8 --{30.0}-> x0 + x7 + x9
  x0 + x13 --{30.0}-> x0 + x12
  x1 + x9 --{30.0}-> x1 + x8
  x1 + x11 + x12 --{30.0}-> x1 + x11 + x13
  x2 + x4 + x6 --{30.0}-> x2 + x4 + x7
  x2 + x11 --{30.0}-> x2 + x10
  x3 + x5 + x10 --{30.0}-> x3 + x5 + x11
  x3 + x7 --{30.0}-> x3 + x6
  x4 + x11 --{30.0}-> x4 + x10
  x5 + x7 --{30.0}-> x5 + x6
  x6 + x6 + x7 --{90.0}-> x6 + x6 + x6
  x6 + x7 + x7 --{90.0}-> x7 + x7 + x7
  x6 + x9 --{30.0}-> x6 + x8
  x8 + x8 + x9 --{90.0}-> x8 + x8 + x8
  x8 + x9 + x9 --{90.0}-> x9 + x9 + x9
  x8 + x12 + x14 --{30.0}-> x8 + x12 + x15
  x9 + x15 --{30.0}-> x9 + x14
  x10 + x10 + x11 --{90.0}-> x10 + x10 + x10
  x10 + x11 + x11 --{90.0}-> x11 + x11 + x11
  x10 + x13 --{30.0}-> x10 + x12
  x12 + x12 + x13 --{90.0}-> x12 + x12 + x12
  x12 + x13 + x13 --{90.0}-> x13 + x13 + x13
  x13 + x15 --{30.0}-> x13 + x14
  x14 + x14 + x15 --{90.0}-> x14 + x14 + x14
  x14 + x15 + x15 --{90.0}-> x15 + x15 + x15
INITIAL CONDITIONS:
  x6(0) = 0.0
  x7(0) = 1.0
  x8(0) = 0.0
  x9(0) = 1.0
  x10(0) = 0.0
  x11(0) = 1.0
  x12(0) = 0.0
  x13(0) = 1.0
  x14(0) = 0.0
  x15(0) = 1.0
```

```
INPUT:
  Tuple(Bool(x0, x1),
        Bool(x2, x3),
        Bool(x4, x5))
OUTPUT:
  Bool(x19, x18)
REACTIONS:
  x0 + x2 + x6 --{30.0}-> x0 + x2 + x7
  x0 + x9 --{30.0}-> x0 + x8
  x1 + x3 + x8 --{30.0}-> x1 + x3 + x9
  x1 + x7 --{30.0}-> x1 + x6
  x2 + x4 + x12 --{30.0}-> x2 + x4 + x13
  x2 + x9 --{30.0}-> x2 + x8
  x2 + x15 --{30.0}-> x2 + x14
  x3 + x5 + x14 --{30.0}-> x3 + x5 + x15
  x3 + x7 --{30.0}-> x3 + x6
  x3 + x13 --{30.0}-> x3 + x12
  x4 + x15 --{30.0}-> x4 + x14
  x5 + x13 --{30.0}-> x5 + x12
  x6 + x6 + x7 --{90.0}-> x6 + x6 + x6
  x6 + x7 + x7 --{90.0}-> x7 + x7 + x7
  x6 + x8 + x10 --{30.0}-> x6 + x8 + x11
  x7 + x11 --{30.0}-> x7 + x10
  x8 + x8 + x9 --{90.0}-> x8 + x8 + x8
  x8 + x9 + x9 --{90.0}-> x9 + x9 + x9
  x9 + x11 --{30.0}-> x9 + x10
  x10 + x10 + x11 --{90.0}-> x10 + x10 + x10
  x10 + x11 + x11 --{90.0}-> x11 + x11 + x11
  x10 + x16 + x18 --{30.0}-> x10 + x16 + x19
  x11 + x19 --{30.0}-> x11 + x18
  x12 + x12 + x13 --{90.0}-> x12 + x12 + x12
  x12 + x13 + x13 --{90.0}-> x13 + x13 + x13
  x12 + x14 + x16 --{30.0}-> x12 + x14 + x17
  x13 + x17 --{30.0}-> x13 + x16
  x14 + x14 + x15 --{90.0}-> x14 + x14 + x14
  x14 + x15 + x15 --{90.0}-> x15 + x15 + x15
  x15 + x17 --{30.0}-> x15 + x16
  x16 + x16 + x17 --{90.0}-> x16 + x16 + x16
  x16 + x17 + x17 --{90.0}-> x17 + x17 + x17
  x17 + x19 --{30.0}-> x17 + x18
  x18 + x18 + x19 --{90.0}-> x18 + x18 + x18
  x18 + x19 + x19 --{90.0}-> x19 + x19 + x19
INITIAL CONDITIONS:
  x6(0)  = 0.0,  x7(0)  = 1.0,  x8(0)  = 0.0,
  x9(0)  = 1.0,  x10(0) = 0.0,  x11(0) = 1.0,
  x12(0) = 0.0,  x13(0) = 1.0,  x14(0) = 0.0,
  x15(0) = 1.0,  x16(0) = 0.0,  x17(0) = 1.0,
  x18(0) = 0.0,  x19(0) = 1.0
```

Fig. 7: Two CRNs compiled using REACTAMOLE. On the left, we have the result of `toCRN unanimousSF`, where `unanimousSF` was constructed using REACTAMOLE's Boolean lifting feature. On the right, we have the result of `toCRN unanimousSF'`, where `unanimousSF'` was manually constructed using REACTAMOLE's built-in signal function combinators. Note that the rate constants of 30 and 90 come from the NAND gate specification of Klinge, Lathrop, and Ellis [15]. These rate constants can theoretically be tuned to an arbitrary precision, but we leave that for future work.

definition u sing t wo X NOR g ates t o v erify pairwise equality and a single AND gate, making sure that both equalities hold true. The resulting CRN for `unanimousSF'` is shown in Figure 7 on the right-hand-side.

As you can see, in this case, the lifted version `unanimousSF` also outperforms the hand-coded version `unanimousSF'`, having fewer reactions (25 versus 35) and species (16 versus 20).

5 Case Study: Amplitude Modulation

We now demonstrate REACTAMOLE's expressiveness via a case study: implementing chemical reaction networks to perform amplitude modulation [27]. 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 -> SF Double Double
constMult d = (constRl d &&& idSF) >>> multSF
```

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

```
carrier :: Double -> SF a Double
carrier w = loop (proj2SF >>> constMult (-w)
  >>> integrateSF 1 >>> constMult w
  >>> integrateSF 0 >>> dupSF)
```

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 `toCRN (carrier w)` would still consist of four species and six reactions.

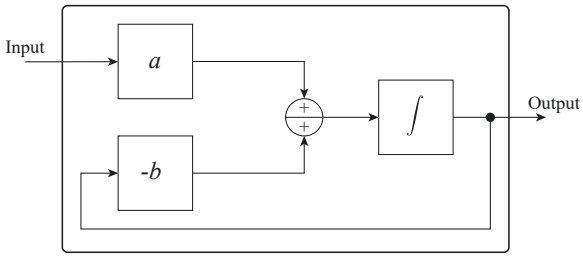

Fig. 8: Visualization of the low-pass filter in REACTAMOLE. The boxes containing constants a and b are constant multipliers.

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 8. 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 [27] can be specified as follows:

```
lowPass :: Double -> Double -> SF Double Double
lowPass a b = loop (first (constMult a)
  >>> second (constMult (-b))
  >>> addSF >>> integrateSF 0 >>> dupSF)
```

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

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

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

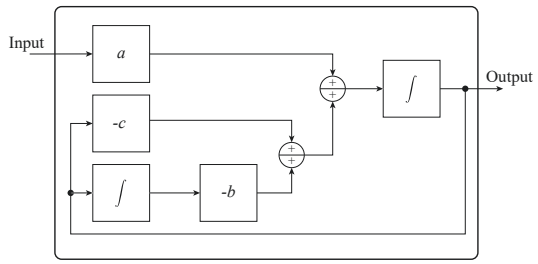

Fig. 9: Visualization of the band-pass filter in REACTAMOLE. The boxes containing constants a, b, and c are constant multipliers.

the band-pass filter presented in [27], which is also included visually in fig. 9.

```
bandPass :: Double -> Double -> Double ->
  SF Double Double
bandPass a b c =
  loop (top *** (mid &&& bot >>> addSF) >>>
        addSF >>> integrateSF 0 >>> dupSF)
  where
    top = constMult a
    mid = constMult (-c)
    bot = integrateSF 0 >>> constMult (-b)
```

Note that `toCRN (bandPass a b c)` consists of the reactions

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

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 [27] that has the medium species M^+ and M^- and can be specified with:

```
modulate :: Double -> SF Double Double
modulate f =
  loop (first (idSF &&& carrier f >>> multSF)
        >>> second negateSF
        >>> addSF >>> integrateSF 0 >>> dupSF)
```

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 [27] also proposed a method to superimpose multiple modulated signals through a single medium $m(t)$, which can be easily accomplished in REACTAMOLE with `addSF :: SF (Double, Double) Double`.

Given a signal $m(t)$ that may carry several modulated signals, we now use the same methods in [27] 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 ->
  SF 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 Abstract Signal Functions

In this section we emphasize that REACTAMOLE is not only capable of specifying *concrete* I/O CRNs, it can also specify *abstract* ones. For example, given a signal $u(t)$, it could be useful to produce a signal $v(t)$ that satisfies $v(t) = \sin(u(t))$ for all t. By differentiating $v(t)$, we can see that it must satisfy the following:

$$\frac{dv}{dt} = \frac{du}{dt} \cdot v, \qquad v(0) = \sin(u(0))$$

Notice that the derivative of v necessarily depends on the derivative of u. As a result, it is impossible

for a *single* CRN to compute the sine of an arbitrary input signal. Nevertheless, it is possible to specify a CRN of $v(t)$ given a *specification* of $u(t)$.

Recall that Reactamole's primitives `addSF` and `multSF` are similar because they also depend on their input's ODEs and initial condition. Because their implementation defers explicit instantiation of their output signal until their input signals are known, they encapsulate a *family* of CRNs. For example, consider the signal function `sqrPlusOne` that computes $x^2 + 1$ of its input.

```
sqrPlusOne :: SF Double Double
sqrPlusOne = constRl 1 &&& sqrSF >>> addSF
  where sqrSF = dupSF >>> multSF
```

Since `sqrPlusOne` depends on the ODEs of its input, the number of species and reactions generated also depends on the signal given as input. Thus, `constRl 3 >>> sqrPlusOne` would produce a CRN with a constant species whose initial concentration is $3^2 + 1 = 10$. However, employing the `sin` signal function from section 3 with `sin >>> sqrPlusOne` would produce the following initial value problem:

```
x0(0) = 0.0, dx0/dt = [+1.0*x1]
x1(0) = 1.0, dx1/dt = [-1.0*x0]
x2(0) = 1.0, dx2/dt = [+2.0*x0*x1]
```

Note that `x0` and `x1` correspond to $sin(t)$ and $cos(t)$, respectively, and upon inspection of the ODE and initial condition for `x2`, we see it indeed has solution $sin(t)^2 + 1$. Moreover, the signal function `sin >>> sqrPlusOne >>> integrateSF 0` produces the ODEs

```
x0(0) = 0.0, dx0/dt = [+1.0*x1]
x1(0) = 1.0, dx1/dt = [-1.0*x0]
x2(0) = 0.0, dx2/dt = [+1.0,+1.0*x0^2]
```

It is important to note that the intermediate variable for $\sin(t)^2 + 1$ was eliminated due to Reactamole's representation of real numbers as polynomials until explicit instantiation is necessary. Now, the variable `x2` has an ODE that consists of $\sin(t)^2 + 1$, exactly as desired.

Many signal functions that cannot be expressed as a single CRN can still be specified in Reactamole. To showcase many of these useful features, we implemented Bournez, Graça, and Pouly's "zoo" of GPAC functions [6]. Reactamole includes templated implementations of exponentiation (`expSF`), sinusoidal functions (`sinSF`, `cosSF`, `tanhSF`), and many more. The `sinSF :: SF Double Double` signal function encapsulates the family of CRNs that compute $\sin(u(t))$. Thus, an equivalent implementation of the `sin` signal we implemented in section 3 is

```
sin = constRl 1 >>> integrateSF 0 >>> sinSF
```

because `sinSF` produces the sine of its input and `constRl 1 >>> integrateSF 0` simply creates a signal that is equal to t.

We can also derive specifications for other common functions using these primitives. For example, the hyperbolic cosine function can be defined using the exponentiation $\cosh(x) = \frac{e^x + e^{-x}}{2}$, which can be implemented easily in Reactamole with

```
coshSF :: SF Double Double
coshSF = expSF &&& (negateSF >>> expSF)
  >>> addSF >>> constMult (1/2)
```

Bournez, Graça, and Pouly also defined approximations of discontinuous functions such as absolute value and sigmoid by using the hyperbolic tangent function. These can be defined in Reactamole with:

```
absSF :: Double -> SF Double Double
absSF d = constMult (1/d) >>> tanhSF &&& idSF
  >>> multSF >>> constAdd d


signSF :: Double -> SF Double Double
signSF d = constMult d >>> tanhSF
```

Both of these signal functions are parameterized with a double d that tunes the accuracy of the approximation. These in turn can be used to define signal functions `maxSF`, `minSF`, and even a rounding function `rndSF`.

7 Related Work

A significant body of past research in the areas of both functional reactive programming and chemical reaction networks [2] laid the foundation for our work. We review relevant areas of this work below, including those that directly influenced Reactamole, as well as other efforts in CRN languages and FRP.

7.1 Chemical Reaction Networks

There are two models of CRNs. The stochastic model represents conditions with a small number of molecules and is probability-based [9, 38, 39]. The deterministic model represents conditions

with substantive chemical concentrations and is modeled by ordinary differential equations. In Reactamole, we have identified that the deterministic model can be realized using functional reactive programming. Identifying a similar framework for the stochastic model remains an open question.

Influences for Reactamole

Reactamole's support for Boolean circuits and functions is based on Ellis *et al.*'s robust NAND gate CRN [15], which is the foundation for Reactamole's logical gates. Klinge *et al.* [28] build upon this work by translating nondeterministic finite automata (NFAs) into I/O CRNs, an important step towards realizing the theoretical potential of this system. The robustness of I/O CRNs inspired us to use them as the basis of Reactamole's typed CRN representation.

While Reactamole uses a novel paradigm for CRNs, it is not the first higher-level language for deterministic CRNs. One important predecessor is CRN++, an imperative language, which utilizes a clock that works similarly to the bits described by Klinge *et al.* [28], with molecular species of oscillating concentrations [40].

Other Linguistical Work

Additional research in the realm of deterministic CRN linguistical support includes CRN compilation into DNA [3, 31] and simulating CRNs [7, 8]. Related linguistical research for molecular programming includes a language for CRN-controlled tile assembly [29], a language for formalizing biological properties using temporal logic and optimizing robustness [7, 18], and other domain specific languages for biochemistry [34].

7.2 Functional Reactive Programming

Overview

There are several helpful overviews of functional reactive programming [41], arrowized FRP [33], and reactive programming more broadly, including many existing languages and approaches [4] that provide insight into the current state of the paradigm. Wan and Hudak also introduce formal semantics for FRP in their paper [41].

Yampa

Reactamole was influenced by several key contributions to the FRP paradigm. The most influential of these is Yampa, another Haskell-embedded FRP DSL, which Reactamole is modeled upon. Yampa uses arrow notation, a generalization of monads, to solve the long-standing FRP issue of preventing time- and space-leaks. This makes it ideal for working with systems under real-time constraints [23]. Reactamole's language structure is inspired by this arrow and signal function design pattern of Yampa, but customized to the domain of CRNs. This arrow construct was first introduced by Hughes in 2000 [24], which marked the creation of arrowized FRP. Yampa supports a newer and more readable syntax for arrows [35], a potential useful addition to Reactamole in the future. Reactamole also uses generalized algebraic data types (GADTs) to ensure that well-typed CRNs correspond to well-formed CRNs, a concept introduced for FRP by Nilsson [32] that was influential both for Reactamole and for Yampa.

Other Implementations

There are also other languages and works of note in the FRP paradigm, with a broad range of applications. These include robotics [36], GUIs [10, 11], and animation [13]. One well-established and respected language within the FRP paradigm is Elm. Unlike other FRP languages, Elm allows the programmer to specify when event orderings can be violated in order to maintain the responsiveness of the UI and improve efficiency. It also increases efficiency by assuming all signal changes occur only after discrete events, and is, therefore, able to sample signals far less frequently [11]. While this approach is quite different from the one we used for Reactamole, it is helpful to show the possibilities of the FRP framework, especially in the early stages of language design. Another language in this domain is Frappé, which predates Elm. Frappé was the first FRP language to be built outside of Haskell, and was implemented in Java, opening the door to wider applications for FRP [10]. One potential future avenue for Reactamole is to move away from Haskell and become a standalone language, as we discuss in section 8.

Analysis of FRP

Beyond programming languages, other important advancements in the FRP realm have been explored, including potential type systems and new strategies for tackling common FRP issues such as time- and space-leaks. Most relevant to Reactamole is the use of Linear-time Temporal Logic (LTL) as a basis for a richer type system than what Haskell provides. LTL is compatible with arrowized FRP and can provide support for expressing temporal language properties [26]. One future direction for Reactamole would be to incorporate LTL into its type system. Another approach to temporal specifications f or F RP is the use of Temporal Stream Logic, which is also compatible with arrowized FRP and has been implemented using the Yampa library [20].

Work has also been conducted on patching time- and space-leaks in FRP languages. A language called Real-Time FRP introduces methods for statically bounding the time and space costs of programs [42]. One method for increasing the efficiency of FRP programs is the Push-Pull method, where values are only recomputed when necessary. This method incorporates data-driven evaluation into the computational approach, greatly reducing the reaction times [14].

8 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 I/O CRNs as signal functions in FRP allows for the efficient construction of a variety of CRNs using relatively few 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 real-valued operations such as addition for CRNs. We also provide a formal construction of the representations of various Haskell types as CRNs, including Booleans, Reals, Pairs, and the `Either` type. All of these features make Reactamole a useful tool for safe, robust, and automated construction and composition of CRNs.

Future Work

There are four main areas for further improvement that we hope to pursue. 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 the `Either` type. Lifting for reals will require some constraints, since these signals cannot be discontinuous. Indeed, the solution of a polynomial system of ODEs is necessarily a real analytic funcion [30].

We would also like to improve the handling of approximations and delays. Currently, `nandSF` and `Either` are approximations because there is some unavoidable delay when working with chemical reactions. Moreover, using Haskell's `Double` type for real values introduces some floating point errors that can affect rate constants and initial conditions of the final CRN. In the future, it would be helpful to build 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 [15] 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, Reactamole is capable of explicitly generating CRNs in a general format, but it stops short of simulating the CRN and plotting the results. We would like to incorporate an ODE solving library and plotting tools for visualization, as well as allowing CRNs to be exported to other tools such as MATLAB SimBiology.

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's features even further to make molecular programming more convenient. 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 [26]. Such a type system can, in turn, enable the efficient au tomatic generation of CRNs from specification, *i.e.*, program synthesis [20].

Acknowledgments. This work was funded in part by National Science Foundation grants 2049911, 1651817, 1545028, and 1900716. The authors also thank the anonymous reviewers for their suggestions and feedback.

References

[1] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, Mar 2006.

[2] Rutherford Aris. Prolegomena to the Rational Analysis of Systems of Chemical Reactions. *Archive for Rational Mechanics and Analysis*, 19(2):81–99, 1964.

[3] 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.

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Comput. Surv.*, 45(4), August 2013.

[5] Olivier Bournez, Daniel S. Graça, and Amaury Pouly. Polynomial time corresponds to solutions of polynomial ordinary differential equations of polynomial length. *J. ACM*, 64(6), oct 2017.

[6] Olivier Bournez, Daniel Graça, and Amaury Pouly. On the functions generated by the general purpose analog computer. *Information and Computation*, 257:34–57, 2017.

[7] Laurence Calzone, François Fages, and Sylvain Soliman. BIOCHAM: an environment for modeling biological systems and formalizing experimental knowledge. *Bioinformatics*, 22(14):1805–1807, 04 2006.

[8] 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.

[9] 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.

[10] Antony Courtney. Frappé: Functional Reactive Programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, Berlin, Heidelberg, 2001. Springer-Verlag.

[11] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 411, Seattle, Washington, USA, 2013. ACM Press.

[12] David Doty, Mahsa Eftekhari, Leszek Gąsieniec, Eric Severson, Przemyslaw Uznański, and Grzegorz Stachowiak. A time and space optimal stable population protocol solving exact majority. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1044–1055, 2022.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the*

second ACM SIGPLAN international conference on Functional programming - ICFP '97, pages 263–273, Amsterdam, The Netherlands, 1997. ACM Press.

[14] 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.

[15] Samuel J. Ellis, Titus H. Klinge, and James I. Lathrop. Robust chemical circuits. *Biosystems*, 186:103983, 2019.

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

[17] 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.

[18] François Fages and Sylvain Soliman. On robustness computation and optimization in BIOCHAM-4. In *Computational Methods in Systems Biology*, pages 292–299, Cham, 2018. Springer International Publishing.

[19] Martin Feinberg. *Foundations of chemical reaction network theory*. Springer, 2019.

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

[21] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 339–347, New York, NY, USA, 2014. Association for Computing Machinery.

[22] Mathieu Hemery, François Fages, and Sylvain Soliman. Compiling elementary mathematical functions into finite chemical reaction networks via a polynomialization algorithm for ODEs. In Eugenio Cinquemani and Loïc Paulevé, editors, *Computational Methods in Systems Biology*, pages 74–90, Cham, 2021. Springer International Publishing.

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

[24] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1):67–111, 2000.

[25] 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.

[26] 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.

[27] Titus H. Klinge and James I. Lathrop. Modulated signals in chemical reaction networks. *CoRR*, abs/2009.06703, 2020.

[28] Titus H. Klinge, James I. Lathrop, and Jack H. Lutz. Robust biomolecular finite automata. *Theoretical Computer Science*, 816:114–143, 2020.

[29] Titus H. Klinge, James I. Lathrop, Sonia Moreno, Hugh D. Potter, Narun K. Raman, and Matthew R. Riley. ALCH: An Imperative Language for Chemical Reaction Network-Controlled Tile Assembly. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing*

and Molecular Programming (DNA 26), volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[30] Steven G Krantz and Harold R Parks. *A primer of real analytic functions*. Springer Science+Business Media, 2002.

[31] Matthew R. Lakin, Simon Youssef, Filippo Polo, Stephen Emmott, and Andrew Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, 2011.

[32] Henrik Nilsson. Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 54–65, New York, NY, USA, 2005. Association for Computing Machinery.

[33] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. Association for Computing Machinery.

[34] Jason Ott, Tyson Loveless, Chris Curtis, Mohsen Lesani, and Philip Brisk. BioScript: programming safe chemistry on laboratories-on-a-chip. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–31, October 2018.

[35] Ross Paterson. A New Notation for Arrows. In *Sigplan Notices - SIGPLAN*, volume 36, pages 229–240, 2001.

[36] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, pages 168–179, New York, NY, USA, 2002. Association for Computing Machinery.

[37] Claude E. Shannon. Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics*, 20(1-4):337–354, 1941.

[38] David Soloveichik. Robust Stochastic Chemical Reaction Networks and Bounded Tau-Leaping. *CoRR*, abs/0803.1030, 2008.

[39] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with Finite Stochastic Chemical Reaction Networks. *Natural Computing*, 7, 2008.

[40] Marko Vasić, David Soloveichik, and Sarfraz Khurshid. CRN++: Molecular programming language. *Natural Computing*, 19(2):391–407, Jun 2020.

[41] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. Association for Computing Machinery.

[42] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 36, 2001.