



# Contextual Equivalence for Signal Flow Graphs

Filippo Bonchi<sup>1</sup>, Robin Piedeleu<sup>2\*</sup>, Paweł Sobociński<sup>3\*\*</sup>, and  
Fabio Zanasi<sup>2\*</sup>(✉)

<sup>1</sup> Università di Pisa, Italy

<sup>2</sup> University College London, UK, {r.piedeleu, f.zanasi}@ucl.ac.uk

<sup>3</sup> Tallinn University of Technology, Estonia

**Abstract.** We extend the signal flow calculus—a compositional account of the classical signal flow graph model of computation—to encompass affine behaviour, and furnish it with a novel operational semantics. The increased expressive power allows us to define a canonical notion of contextual equivalence, which we show to coincide with denotational equality. Finally, we characterise the realisable fragment of the calculus: those terms that express the computations of (affine) signal flow graphs.

**Keywords:** signal flow graphs · affine relations · full abstraction · contextual equivalence · string diagrams

## 1 Introduction

Compositional accounts of models of computation often lead one to consider *relational* models because a decomposition of an input-output system might consist of internal parts where flow and causality are not always easy to assign. These insights led Willems [33] to introduce a new current of control theory, called *behavioural* control: roughly speaking, behaviours and observations are of prime concern, notions such as state, inputs or outputs are secondary. Independently, programming language theory converged on similar ideas, with *contextual equivalence* [25,28] often considered as *the* equivalence: programs are judged to be different if we can find some context in which one behaves differently from the other, and what is observed about “behaviour” is often something quite canonical and simple, such as termination. Hoare [17] and Milner [23] discovered that these programming language theory innovations also bore fruit in the non-deterministic context of concurrency. Here again, research converged on studying simple and canonical contextual equivalences [24,18].

This paper brings together all of the above threads. The model of computation of interest for us is that of signal flow graphs [32,21], which are feedback systems well known in control theory [21] and widely used in the modelling of linear dynamical systems (in continuous time) and signal processing circuits (in

\* Supported by EPSRC grant EP/R020604/1.

\*\* Supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001)

discrete time). The *signal flow calculus* [10,9] is a syntactic presentation with an underlying compositional denotational semantics in terms of linear relations. Armed with *string diagrams* [31] as a syntax, the tools and concepts of programming language theory and concurrency theory can be put to work and the calculus can be equipped with a structural operational semantics. However, while in previous work [9] a connection was made between operational equivalence (essentially trace equivalence) and denotational equality, the signal flow calculus was not quite expressive enough for contextual equivalence to be a useful notion.

The crucial step turns out to be moving from *linear* relations to *affine* relations, i.e. linear subspaces translated by a vector. In recent work [6], we showed that they can be used to study important physical phenomena, such as current and voltage sources in electrical engineering, as well as fundamental synchronisation primitives in concurrency, such as mutual exclusion. Here we show that, in addition to yielding compelling mathematical domains, affinity proves to be the magic ingredient that ties the different components of the story of signal flow graphs together: it provides us with a canonical and simple notion of observation to use for the *definition* of contextual equivalence, and gives us the expressive power to prove a bona fide full abstraction result that relates contextual equivalence with denotational equality.

To obtain the above result, we extend the signal flow calculus to handle affine behaviour. While the denotational semantics and axiomatic theory appeared in [6], the operational account appears here for the first time and requires some technical innovations: instead of traces, we consider *trajectories*, which are infinite traces that may start in the past. To record the time, states of our transition system have a runtime environment that keeps track of the global clock.

Because the affine signal flow calculus is oblivious to flow directionality, some terms exhibit pathological operational behaviour. We illustrate these phenomena with several examples. Nevertheless, for the linear sub-calculus, it is known [9] that every term is denotationally equal to an executable realisation: one that is in a form where a consistent flow can be identified, like the classical notion of signal flow graph. We show that the question has a more subtle answer in the affine extension: not all terms are realisable as (affine) signal flow graphs. However, we are able to characterise the class of diagrams for which this is true.

*Related work.* Several authors studied signal flow graphs by exploiting concepts and techniques of programming language semantics, see e.g. [4,22,29,2]. The most relevant for this paper is [2], which, independently from [10], proposed the same syntax and axiomatisation for the ordinary signal flow calculus and shares with our contribution the same methodology: the use of *string diagrams* as a mathematical playground for the compositional study of different sorts of systems. The idea is common to diverse, cross-disciplinary research programmes, including Categorical Quantum Mechanics [1,11,12], Categorical Network Theory [3], Monoidal Computer [26,27] and the analysis of (a)synchronous circuits [14,15].

*Outline* In Section 2 we recall the affine signal flow calculus. Section 3 introduces the operational semantics for the calculus. Section 4 defines contextual equivalence and proves full abstraction. Section 5 introduces a well-behaved class of

circuits, that denotes functional input-output systems, laying the groundwork for Section 6, in which the concept of realisability is introduced before a characterisation of which circuit diagrams are realisable. Missing proofs can be found in the extended version of this paper [7].

## 2 Background: the Affine Signal Flow Calculus

The *Affine Signal Flow Calculus* extends the signal flow calculus [9] with an extra generator  $\dashv$  that allows to express affine relations. In this section, we first recall its syntax and denotational semantics from [6] and then we highlight two key properties for proving full abstraction that are enabled by the affine extension. The operational semantics is delayed to the next section.

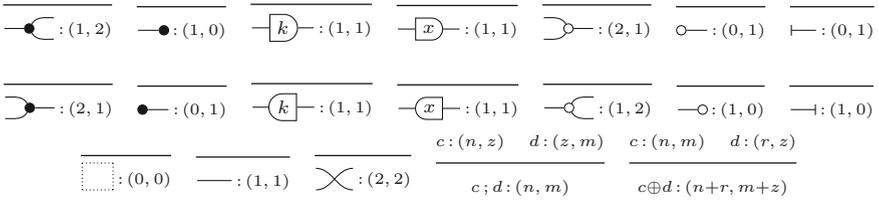


Fig. 1. Sort inference rules.

### 2.1 Syntax

$$c ::= \bullet \mid \bullet \dashv \mid \boxed{k} \dashv \mid \dashv \boxed{x} \mid \dashv \dashv \mid \dashv \dashv \mid \dashv \dashv \mid \dashv \dashv \quad (1)$$

$$\bullet \dashv \mid \dashv \bullet \mid \dashv \boxed{k} \mid \dashv \dashv \boxed{x} \mid \dashv \dashv \dashv \mid \dashv \dashv \dashv \mid \dashv \dashv \dashv \quad (2)$$

$$\boxed{\phantom{x}} \mid \dashv \mid \dashv \dashv \mid c \oplus c \mid c ; c \quad (3)$$

The syntax of the calculus, generated by the grammar above, is parametrised over a given field  $k$ , with  $k$  ranging over  $\mathbb{k}$ . We refer to the constants in rows (1)–(2) as *generators*. Terms are constructed from generators,  $\boxed{\phantom{x}}$ ,  $\dashv$ ,  $\dashv \dashv$ , and the two binary operations in (3). We will only consider those terms that are *sortable*, i.e. they can be associated with a pair  $(n, m)$ , with  $n, m \in \mathbb{N}$ . Sortable terms are called *circuits*: intuitively, a circuit with sort  $(n, m)$  has  $n$  ports on the left and  $m$  on the right. The sorting discipline is given in Fig. 1. We delay discussion of computational intuitions to Section 3 but, for the time being, we observe that the generators of row (2) are those of row (1) “reflected about the  $y$ -axis”.

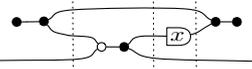
### 2.2 String Diagrams

It is convenient to consider circuits as the arrows of a symmetric monoidal category  $\text{ACirc}$  (for Affine Circuits). Objects of  $\text{ACirc}$  are natural numbers (thus

ACirc is a *prop* [19]) and morphisms  $n \rightarrow m$  are the circuits of sort  $(n, m)$ , quotiented by the laws of symmetric monoidal categories [20,31]<sup>4</sup>. The circuit grammar yields the symmetric monoidal structure of ACirc: sequential composition is given by  $c; d$ , the monoidal product is given by  $c \oplus d$ , and identities and symmetries are built by pasting together  $\text{---}$  and  $\text{---}$  in the obvious way. We will adopt the usual convention of writing morphisms of ACirc as *string diagrams*,

meaning that  $c; c'$  is drawn  $\text{---} \boxed{c} \text{---} \boxed{c'} \text{---}$  and  $c \oplus c'$  is drawn  $\text{---} \begin{array}{|c|} \hline c \\ \hline c' \\ \hline \end{array} \text{---}$ . More suc-

cinctly, ACirc is the free prop on generators (1)-(2). The free prop on (1)-(2) sans  $\text{---}$  and  $\text{---}$ , hereafter called Circ, is the signal flow calculus from [9].

*Example 1.* The diagram  represents the circuit

$$((\bullet \text{---}; \bullet \text{---}) \oplus \text{---}); (\text{---} \oplus (\text{---} \text{---}); \bullet \text{---}); (((\text{---} \oplus \text{---}) \oplus \text{---}) \oplus \text{---}); ((\bullet \text{---}; \bullet \text{---}) \oplus \text{---})).$$

### 2.3 Denotational Semantics and Axiomatisation

The semantics of circuits can be given denotationally by means of affine relations.

**Definition 1.** Let  $k$  be a field. An affine subspace of  $k^d$  is a subset  $V \subseteq k^d$  that is either empty or for which there exists a vector  $a \in k^d$  and a linear subspace  $L$  of  $k^d$  such that  $V = \{a + v \mid v \in L\}$ . A  $k$ -affine relation of type  $n \rightarrow m$  is an affine subspace of  $k^n \times k^m$ , considered as a  $k$ -vector space.

Note that every linear subspace is affine, taking  $a$  above to be the zero vector. Affine relations can be organised into a prop:

**Definition 2.** Let  $k$  be a field. Let  $\text{ARel}_k$  be the following prop:

- arrows  $n \rightarrow m$  are  $k$ -affine relations.
- composition is relational: given  $G = \{(u, v) \mid u \in k^n, v \in k^m\}$  and  $H = \{(v, w) \mid v \in k^m, w \in k^l\}$ , their composition is  $G; H := \{(u, w) \mid \exists v. (u, v) \in G \wedge (v, w) \in H\}$ .
- monoidal product given by  $G \oplus H = \left\{ \left( \begin{pmatrix} u \\ u' \end{pmatrix}, \begin{pmatrix} v \\ v' \end{pmatrix} \right) \mid (u, v) \in G, (u', v') \in H \right\}$ .

In order to give semantics to ACirc, we use the prop of affine relations over the field  $k(x)$  of fractions of polynomials in  $x$  with coefficients from  $k$ . Elements  $q \in k(x)$  are a fractions  $\frac{k_0 + k_1 \cdot x^1 + k_2 \cdot x^2 + \dots + k_n \cdot x^n}{l_0 + l_1 \cdot x^1 + l_2 \cdot x^2 + \dots + l_m \cdot x^m}$  for some  $n, m \in \mathbb{N}$  and  $k_i, l_i \in k$ . Sum, product, 0 and 1 in  $k(x)$  are defined as usual.

<sup>4</sup> This quotient is harmless: both the denotational semantics from [6] and the operational semantics we introduce in this paper satisfy those axioms on the nose.

**Definition 3.** The prop morphism  $\llbracket \cdot \rrbracket : \text{ACirc} \rightarrow \text{ARel}_{\mathbf{k}(x)}$  is inductively defined on circuits as follows. For the generators in (1)

$$\begin{aligned} \text{---} \curvearrowright &\mapsto \left\{ \left( p, \begin{pmatrix} p \\ p \end{pmatrix} \right) \mid p \in \mathbf{k}(x) \right\} & \curvearrowleft \text{---} &\mapsto \left\{ \left( \begin{pmatrix} p \\ q \end{pmatrix}, p+q \right) \mid p, q \in \mathbf{k}(x) \right\} \\ \text{---} \bullet &\mapsto \{ (p, \bullet) \mid p \in \mathbf{k}(x) \} & \circ \text{---} &\mapsto \{ (\bullet, 0) \} & \text{---} &\mapsto \{ (\bullet, 1) \} \\ \text{---} \boxed{r} \text{---} &\mapsto \{ (p, p \cdot r) \mid p \in \mathbf{k}(x) \} & \text{---} \boxed{x} \text{---} &\mapsto \{ (p, p \cdot x) \mid p \in \mathbf{k}(x) \} \end{aligned}$$

where  $\bullet$  is the only element of  $\mathbf{k}(x)^0$ . The semantics of components in (2) is symmetric, e.g.  $\bullet \text{---}$  is mapped to  $\{ (p, \bullet) \mid p \in \mathbf{k}(x) \}$ . For (3)

$$\begin{aligned} \text{---} &\mapsto \{ (p, p) \mid p \in \mathbf{k}(x) \} & \times &\mapsto \left\{ \left( \begin{pmatrix} p \\ q \end{pmatrix}, \begin{pmatrix} q \\ p \end{pmatrix} \right) \mid p, q \in \mathbf{k}(x) \right\} \\ \square &\mapsto \{ (\bullet, \bullet) \} & c_1 \oplus c_2 &\mapsto \llbracket c_1 \rrbracket \oplus \llbracket c_2 \rrbracket & c_1 ; c_2 &\mapsto \llbracket c_1 \rrbracket ; \llbracket c_2 \rrbracket \end{aligned}$$

The reader can easily check that the pair of 1-dimensional vectors  $\left(1, \frac{1}{1-x}\right) \in \mathbf{k}(x)^1 \times \mathbf{k}(x)^1$  belongs to the denotation of the circuit in Example 1.

The denotational semantics enjoys a sound and complete axiomatisation. The axioms involve only basic interactions between the generators (1)-(2). The resulting theory is that of *Affine Interacting Hopf Algebras* ( $\text{aIH}$ ). The generators in (1) form a Hopf algebra, those in (2) form another Hopf algebra, and the interaction of the two give rise to two Frobenius algebras. We refer the reader to [6] for the full set of equations and all further details.

**Proposition 1.** For all  $c, d$  in  $\text{ACirc}$ ,  $\llbracket c \rrbracket = \llbracket d \rrbracket$  if and only if  $c \stackrel{\text{aIH}}{=} d$ .

## 2.4 Affine vs Linear Circuits

It is important to highlight the differences between  $\text{ACirc}$  and  $\text{Circ}$ . The latter is the purely linear fragment: circuit diagrams of  $\text{Circ}$  denote exactly the *linear* relations over  $\mathbf{k}(x)$  [8], while those of  $\text{ACirc}$  denote the *affine* relations over  $\mathbf{k}(x)$ .

The additional expressivity afforded by affine circuits is essential for our development. One crucial property is that every polynomial fraction can be expressed as an affine circuit of sort  $(0, 1)$ .

**Lemma 1.** For all  $p \in \mathbf{k}(x)$ , there is  $c_p \in \text{ACirc}[0, 1]$  with  $\llbracket c_p \rrbracket = \{ (\bullet, p) \}$ .

*Proof.* For each  $p \in \mathbf{k}(x)$ , let  $P$  be the linear subspace generated by the pair of 1-dimensional vectors  $(1, p)$ . By fullness of the denotational semantics of  $\text{Circ}$  [8], there exists a circuit  $c$  in  $\text{Circ}$  such that  $\llbracket c \rrbracket = P$ . Then,  $\llbracket \text{---} ; c \rrbracket = \{ (\bullet, p) \}$ .  $\square$

The above observation yields the following:

**Proposition 2.** Let  $(u, v) \in \mathbf{k}(x)^n \times \mathbf{k}(x)^m$ . There exist circuits  $c_u \in \text{ACirc}[0, n]$  and  $c_v \in \text{ACirc}[m, 0]$  such that  $\llbracket c_u \rrbracket = \{ (\bullet, u) \}$  and  $\llbracket c_v \rrbracket = \{ (v, \bullet) \}$ .

*Proof.* Let  $u = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}$  and  $v = \begin{pmatrix} q_1 \\ \vdots \\ q_m \end{pmatrix}$ . By Lemma 1, for each  $p_i$ , there exists a circuit  $c_{p_i}$  such that  $\llbracket c_{p_i} \rrbracket = \{(\bullet, p_i)\}$ . Let  $c_u = c_{p_1} \oplus \dots \oplus c_{p_n}$ . Then  $\llbracket c_u \rrbracket = \{(\bullet, u)\}$ . For  $c_v$ , it is enough to see that Proposition 1 also holds with 0 and 1 switched, then use the argument above.  $\square$

Proposition 2 asserts that any behaviour  $(u, v)$  occurring in the denotation of some circuit  $c$ , i.e., such that  $(u, v) \in \llbracket c \rrbracket$ , can be expressed by a pair of circuits  $(c_u, c_v)$ . We will, in due course, think of such a pair as a *context*, namely an environment with which a circuit can interact. Observe that this is not possible with the linear fragment  $\text{Circ}$ , since the only singleton linear subspace is 0.

Another difference between linear and affine concerns circuits of sort  $(0, 0)$ . Indeed  $k(x)^0 = \{\bullet\}$ , and the only linear relation over  $k(x)^0 \times k(x)^0$  is the singleton  $\{(\bullet, \bullet)\}$ , which is  $id_0$  in  $\text{ARel}_{k(x)}$ . But there is another affine relation, namely the *empty relation*  $\emptyset \in k(x)^0 \times k(x)^0$ . This can be represented by  $\dashv\circ$ , for instance, since  $\llbracket \dashv\circ \rrbracket = \{(\bullet, 1)\}; \{(0, \bullet)\} = \emptyset$ .

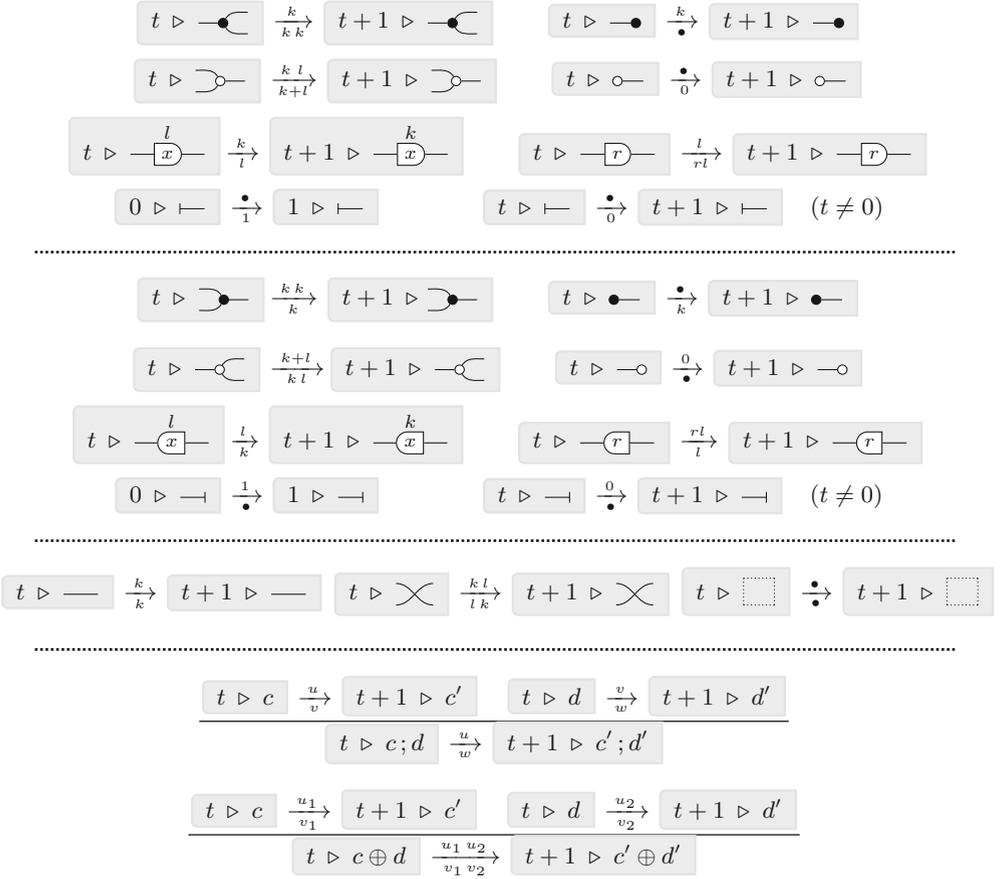
**Proposition 3.** *Let  $c \in \text{ACirc}[0, 0]$ . Then  $\llbracket c \rrbracket$  is either  $id_0$  or  $\emptyset$ .*

### 3 Operational Semantics for Affine Circuits

Here we give the structural operational semantics of affine circuits, building on previous work [9] that considered only the core linear fragment,  $\text{Circ}$ . We consider circuits to be *programs* that have an observable behaviour. Observations are possible interactions at the circuit’s interface. Since there are two interfaces: a left and a right, each transition has two labels.

In a transition  $t \triangleright c \xrightarrow[v]{u} t' \triangleright c'$ ,  $c$  and  $c'$  are *states*, that is, circuits augmented with information about which values  $k \in k$  are stored in each register  $(\boxed{x})$  and  $(\boxed{x})$  at that instant of the computation. When transitioning to  $c'$ , the  $v$  above the arrow is a vector of values with which  $c$  synchronises on the left, and the  $w$  below the arrow accounts for the synchronisation on the right. States are decorated with runtime contexts:  $t$  and  $t'$  are (possibly negative) integers that—intuitively—indicate the time when the transition happens. Indeed, in Fig. 2, every rule advances time by 1 unit. “Negative time” is important: as we shall see in Example 3, some executions must start in the past.

The rules in the top section of Fig. 2 provide the semantics for the generators in (1):  $\bullet$  is a *copier*, duplicating the signal arriving on the left;  $\bullet$  accepts any signal on the left and discards it, producing nothing on the right;  $\cup$  is an *adder* that takes two signals on the left and emits their sum on the right,  $\circ$  emits the constant 0 signal on the right;  $\boxed{k}$  is an *amplifier*, multiplying the signal on the left by the scalar  $k \in k$ . All the generators described so far are stateless. State is provided by  $\boxed{l}$  which is a *register*; a synchronous one place buffer with the value  $l$  stored. When it receives some value  $k$  on the left, it emits  $l$  on the right and stores  $k$ . The behaviour of the affine generator  $\dashv\circ$



**Fig. 2.** Structural rules for operational semantics, with  $p \in \mathbb{Z}$ ,  $k, l$  ranging over  $\mathbb{k}$  and  $u, v, w$  vectors of elements of  $\mathbb{k}$  of the appropriate size. The only vector of  $\mathbb{k}^0$  is written as  $\bullet$  (as in Definition 3), while a vector  $(k_1 \dots k_n)^T \in \mathbb{k}^n$  as  $k_1 \dots k_n$ .

depends on the time: when  $t = 0$ , it emits 1, otherwise it emits 0. Observe that the behaviour of all other generators is time-independent.

So far, we described the behaviour of the components in (1) using the intuition that signal flows from left to right: in a transition  $\frac{v}{w} \rightarrow$ , the signal  $v$  on the left is thought as trigger and  $w$  as effect. For the generators in (2), whose behaviour is defined by the rules in the second section of Fig. 2, the behaviour is symmetric—indeed, here it is helpful to think of signals as flowing from right to left. The next section of Fig. 2 specifies the behaviours of the structural connectors of (3):  $\bowtie$  is a *twist*, swapping two signals,  $\square$  is the empty circuit and  $\text{—}$  is the *identity* wire: the signals on the left and on the right ports are equal. Finally, the rule for sequential ; composition forces the two components to have the same value  $v$  on the shared interface, while for parallel  $\oplus$  composition,

components can proceed independently. Observe that both forms of composition require component transitions to happen at the same time.

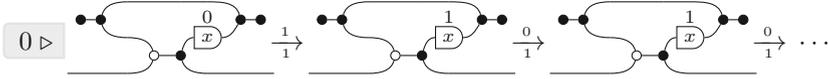
**Definition 4.** Let  $c \in \text{ACirc}$ . The initial state  $c_0$  of  $c$  is the one where all the registers store 0. A computation of  $c$  starting at time  $t \leq 0$  is a (possibly infinite) sequence of transitions

$$t \triangleright c_0 \xrightarrow[w_t]{v_t} t+1 \triangleright c_1 \xrightarrow[w_{t+1}]{v_{t+1}} t+2 \triangleright c_2 \xrightarrow[w_{t+2}]{v_{t+2}} \dots \quad (4)$$

Since all transitions increment the time by 1, it suffices to record the time at which a computation starts. As a result, to simplify notation, we will omit the runtime context after the first transition and, instead of (4), write

$$t \triangleright c_0 \xrightarrow[w_t]{v_t} c_1 \xrightarrow[w_{t+1}]{v_{t+1}} c_2 \xrightarrow[w_{t+2}]{v_{t+2}} \dots$$

*Example 2.* The circuit in Example 1 can perform the following computation.



In the example above, the flow has a clear left-to-right orientation, albeit with a feedback loop. For arbitrary circuits of  $\text{ACirc}$  this is not always the case, which sometimes results in unexpected operational behaviour.

*Example 3.* In  $\vdash \boxed{x}$  is not possible to identify a consistent flow:  $\vdash$  goes from left to right, while  $\boxed{x}$  from right to left. Observe that there is no computation starting at  $t = 0$ , since in the initial state the register contains 0 while  $\vdash$  must emit 1. There is, however, a (unique!) computation starting at time  $t = -1$ , that loads the register with 1 before  $\vdash$  can also emit 1 at time  $t = 0$ .

$$-1 \triangleright \vdash \boxed{x} \xrightarrow[1]{0} \vdash \boxed{x} \xrightarrow[0]{1} \vdash \boxed{x} \xrightarrow[0]{0} \vdash \boxed{x} \xrightarrow[0]{0} \dots$$

Similarly,  $\vdash \boxed{x} \boxed{x}$  features a unique computation starting at time  $t = -2$ .

$$-2 \triangleright \vdash \boxed{x} \boxed{x} \xrightarrow[1]{0} \vdash \boxed{x} \boxed{x} \xrightarrow[0]{1} \vdash \boxed{x} \boxed{x} \xrightarrow[0]{0} \vdash \boxed{x} \boxed{x} \xrightarrow[0]{0} \dots$$

It is worthwhile clarifying the reason why, in the affine calculus, some computations start in the past. As we have already mentioned, in the linear fragment the semantics of all generators is time-independent. It follows easily that time-independence is a property enjoyed by all purely linear circuits. The behaviour of  $\vdash$ , however, enforces a particular action to occur at time 0. Considering this in conjunction with a right-to-left register results in  $\vdash \boxed{x}$ , and the effect is to anticipate that action by one step to time -1, as shown in Example 3. It is obvious that this construction can be iterated, and it follows that the presence of a single time-dependent generator results in a calculus in which the computation of some terms must start at a finite, but unbounded time in the past.

*Example 4.* Another circuit with conflicting flow is  $\vdash\circ$ . Here there is no possible transition at  $t = 0$ , since at that time  $\vdash$  must emit a 1 and  $\circ$  can only synchronise on a 0. Instead, the circuit  $\boxed{\phantom{x}}$  can always perform an infinite computation  $t \triangleright \boxed{\phantom{x}} \xrightarrow{\bullet} \boxed{\phantom{x}} \xrightarrow{\bullet} \dots$ , for any  $t \leq 0$ . Roughly speaking, the computations of these two  $(0, 0)$  circuits are operational mirror images of the two possible denotations of Proposition 3. This intuition will be made formal in Section 4. For now, it is worth observing that for all  $c$ ,  $\boxed{\phantom{x}} \oplus c$  can perform the same computations of  $c$ , while  $\vdash\circ \oplus c$  cannot ever make a transition at time 0.

*Example 5.* Consider the circuit  $\overline{\text{---}(\text{---}x\text{---})\text{---}}$ , which again features conflicting flow. Our equational theory equates it with  $\text{---}$ , but the computations involved are subtly different. Indeed, for any sequence  $a_i \in \mathbf{k}$ , it is obvious that  $\text{---}$  admits the computation

$$0 \triangleright \text{---} \xrightarrow{\frac{a_0}{a_0}} \text{---} \xrightarrow{\frac{a_1}{a_1}} \text{---} \xrightarrow{\frac{a_2}{a_2}} \dots \quad (5)$$

The circuit  $\overline{\text{---}(\text{---}x\text{---})\text{---}}$  admits a similar computation, but we must begin at time  $t = -1$  in order to first “load” the registers with  $a_0$ :

$$-1 \triangleright \overline{\text{---}(\text{---}x\text{---})\text{---}} \xrightarrow{\frac{0}{0}} \overline{\text{---}(\text{---}x\text{---})\text{---}} \xrightarrow{\frac{a_0}{a_0}} \overline{\text{---}(\text{---}x\text{---})\text{---}} \xrightarrow{\frac{a_1}{a_1}} \overline{\text{---}(\text{---}x\text{---})\text{---}} \xrightarrow{\frac{a_2}{a_2}} \dots \quad (6)$$

The circuit  $\overline{\text{---}(\text{---}x\text{---})\text{---}}$ , which again is equated with  $\text{---}$  by the equational theory, is more tricky. Although every computation of  $\text{---}$  can be reproduced,  $\overline{\text{---}(\text{---}x\text{---})\text{---}}$  admits additional, problematic computations. Indeed, consider

$$0 \triangleright \overline{\text{---}(\text{---}x\text{---})\text{---}} \xrightarrow{\frac{0}{1}} \overline{\text{---}(\text{---}x\text{---})\text{---}} \quad (7)$$

at which point no further transition is possible—the circuit can deadlock.

The following lemma is an easy consequence of the rules of Fig. 2 and follows by structural induction. It states that all circuits can stay idle *in the past*.

**Lemma 2.** *Let  $c \in \text{ACirc}[n, m]$  with initial state  $c_0$ . Then  $t \triangleright c_0 \xrightarrow{0} c_0$  if  $t < 0$ .*

### 3.1 Trajectories

For the non-affine version of the signal flow calculus, we studied in [9] *traces* arising from computations. For the affine extension, this is not possible since, as explained above, we must also consider computations that start in the past. In this paper, rather than traces we adopt a common control theoretic notion.

**Definition 5.** *An  $(n, m)$ -trajectory  $\sigma$  is a  $\mathbb{Z}$ -indexed sequence  $\sigma : \mathbb{Z} \rightarrow \mathbf{k}^n \times \mathbf{k}^m$  that is finite in the past, i.e., for which  $\exists j \in \mathbb{Z}$  such that  $\sigma(i) = (0, 0)$  for  $i \leq j$ .*

By the universal property of the product we can identify  $\sigma : \mathbb{Z} \rightarrow \mathbf{k}^n \times \mathbf{k}^m$  with the pairing  $\langle \sigma_l, \sigma_r \rangle$  of  $\sigma_l : \mathbb{Z} \rightarrow \mathbf{k}^n$  and  $\sigma_r : \mathbb{Z} \rightarrow \mathbf{k}^m$ . A  $(k, m)$ -trajectory  $\sigma$  and  $(m, n)$ -trajectory  $\tau$  are *compatible* if  $\sigma_r = \tau_l$ . In this case, we can define

their composite, a  $(k, n)$ -trajectory  $\sigma; \tau$  by  $\sigma; \tau := \langle \sigma_l, \tau_r \rangle$ . Given an  $(n_1, m_1)$ -trajectory  $\sigma_1$ , and an  $(n_2, m_2)$ -trajectory  $\sigma_2$ , their product, an  $(n_1+n_2, m_1+m_2)$ -trajectory  $\sigma_1 \oplus \sigma_2$ , is defined  $(\sigma_1 \oplus \sigma_2)(i) := \begin{pmatrix} \sigma(i) \\ \tau(i) \end{pmatrix}$ . Using these two operations we can organise *sets* of trajectories into a prop.

**Definition 6.** *The composition of two sets of trajectories is defined as  $S; T := \{\sigma; \tau \mid \sigma \in S, \tau \in T \text{ are compatible}\}$ . The product of sets of trajectories is defined as  $S_1 \oplus S_2 := \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in S_1, \sigma_2 \in S_2\}$ .*

Clearly both operations are strictly associative. The unit for  $\oplus$  is the singleton with the unique  $(0, 0)$ -trajectory. Also  $;$  has a two sided identity, given by sets of “copycat”  $(n, n)$ -trajectories. Indeed, we have that:

**Proposition 4.** *Sets of  $(n, m)$ -trajectories are the arrows  $n \rightarrow m$  of a prop  $\text{Traj}$  with composition and monoidal product given as in Definition 6.*

$\text{Traj}$  serves for us as the domain for operational semantics: given a circuit  $c$  and an *infinite* computation

$$t \triangleright c_0 \xrightarrow[u_t]{u_t} c_1 \xrightarrow[v_{t+1}]{u_{t+1}} c_2 \xrightarrow[v_{t+2}]{u_{t+2}} \dots$$

its associated trajectory  $\sigma$  is

$$\sigma(i) = \begin{cases} (u_i, v_i) & \text{if } i \geq t, \\ (0, 0) & \text{otherwise.} \end{cases} \quad (8)$$

**Definition 7.** *For a circuit  $c$ ,  $\langle c \rangle$  is the set of trajectories given by its infinite computations, following the translation (8) above.*

The assignment  $c \mapsto \langle c \rangle$  is compositional, that is:

**Theorem 1.**  $\langle \cdot \rangle: \text{ACirc} \rightarrow \text{Traj}$  is a morphism of props.

*Example 6.* Consider the computations (5) and (6) from Example 5. According to (8) both are translated into the trajectory  $\sigma$  mapping  $i \geq 0$  into  $(a_i, a_i)$  and  $i < 0$  into  $(0, 0)$ . The reader can easily verify that, more generally, it holds that  $\langle \text{---} \rangle = \langle \text{---} \langle \text{---} \rangle \text{---} \rangle$ . At this point it is worth to remark that the two circuits would be distinguished when looking at their traces: the trace of computation (5) is different from the trace of (6). Indeed, the full abstraction result in [9] does not hold for all circuits, but only for those of a certain kind. The affine extension obliges us to consider computations that starts in the past and, in turn, this drives us toward a stronger full abstraction result, shown in the next section.

Before concluding, it is important to emphasise that  $\langle \text{---} \rangle = \langle \text{---} \langle \text{---} \rangle \text{---} \rangle$  also holds. Indeed, problematic computations, like (7), are all finite and, by definition, do not give rise to any trajectory. The reader should note that the use of trajectories is not a semantic device to get rid of problematic computations. In fact, trajectories do not appear in the statement of our full abstraction result; they are merely a convenient tool to prove it. Another result (Proposition 9) independently takes care of ruling out problematic computations.

## 4 Contextual Equivalence and Full Abstraction

This section contains the main contribution of the paper: a traditional full abstraction result asserting that contextual equivalence agrees with denotational equivalence. It is not a coincidence that we prove this result in the affine setting: affinity plays a crucial role, both in its statement and proof. In particular, Proposition 3 gives us two possibilities for the denotation of  $(0, 0)$  circuits: (i)  $\emptyset$ —which, roughly speaking, means that there is a problem (see e.g. Example 4) and no infinite computation is possible—or (ii)  $id_0$ , in which case infinite computations are possible. This provides us with a basic notion of observation, akin to observing termination vs non-termination in the  $\lambda$ -calculus.

**Definition 8.** For a circuit  $c \in \text{ACirc}[0, 0]$  we write  $c \uparrow$  if  $c$  can perform an infinite computation and  $c \not\uparrow$  otherwise. For instance  $\square \uparrow$ , while  $\dashv \circ \not\uparrow$ .

To be able to make observations about arbitrary circuits we need to introduce an appropriate notion of context. Roughly speaking, contexts for us are  $(0, 0)$ -circuits with a hole into which we can plug another circuit. Since ours is a variable-free presentation, “dangling wires” assume the role of free variables [16]: restricting to  $(0, 0)$  contexts is therefore analogous to considering *ground* contexts—i.e. contexts with no free variables—a standard concept of programming language theory.

To define contexts formally, we extend the syntax of Section 2.1 with an extra generator “ $-$ ” of sort  $(n, m)$ . A  $(0, 0)$ -circuit of this extended syntax is a *context* when “ $-$ ” occurs exactly once. Given an  $(n, m)$ -circuit  $c$  and a context  $C[-]$ , we write  $C[c]$  for the circuit obtained by replacing the unique occurrence of “ $-$ ” by  $c$ .

With this setup, given an  $(n, m)$ -circuit  $c$ , we can insert it into a context  $C[-]$  and observe the possible outcome: either  $C[c] \uparrow$  or  $C[c] \not\uparrow$ . This naturally leads us to contextual equivalence and the statement of our main result.

**Definition 9.** Given  $c, d \in \text{ACirc}[n, m]$ , we say that they are contextually equivalent, written  $c \equiv d$ , if for all contexts  $C[-]$ ,

$$C[c] \uparrow \text{ iff } C[d] \uparrow .$$

*Example 7.* Recall from Example 5, the circuits  $\text{---}$  and  $\text{---} \square \text{---}$ . Take the context  $C[-] = c_\sigma ; - ; c_\tau$  for  $c_\sigma \in \text{ACirc}[0, 1]$  and  $c_\tau \in \text{ACirc}[1, 0]$ . Assume that  $c_\sigma$  and  $c_\tau$  have a single infinite computation. Call  $\sigma$  and  $\tau$  the corresponding trajectories. If  $\sigma = \tau$ , both  $C[\text{---}]$  and  $C[\text{---} \square \text{---}]$  would be able to perform an infinite computation. Instead if  $\sigma \neq \tau$ , none of them would perform any infinite computation:  $\text{---}$  would stop at time  $t$ , for  $t$  the first moment such that  $\sigma(t) \neq \tau(t)$ , while  $C[\text{---} \square \text{---}]$  would stop at time  $t + 1$ .

Now take as context  $C[-] = \bullet \text{---} ; - ; \text{---} \bullet$ . In contrast to  $c_\sigma$  and  $c_\tau$ ,  $\bullet \text{---}$  and  $\text{---} \bullet$  can perform more than one single computation: at any time they can nondeterministically emit any value. Thus every computation of  $C[\text{---}] = \bullet \text{---}$

can *always* be extended to an infinite one, forcing synchronisation of  $\bullet-$  and  $- \bullet$  at each step. For  $C[-\boxed{x}-\boxed{x}-] = \bullet-\boxed{x}-\boxed{x}-\bullet$ ,  $\bullet-$  and  $- \bullet$  may emit different values at time  $t$ , but the computation will get stuck at  $t + 1$ . However, our definition of  $\uparrow$  only cares about whether  $C[-\boxed{x}-\boxed{x}-]$  can perform an infinite computation. Indeed it can, as long as  $\bullet-$  and  $- \bullet$  consistently emit the same value at each time step.

If we think of contexts as tests, and say that a circuit  $c$  passes test  $C[-]$  if  $C[c]$  perform an infinite computation, then our notion of contextual equivalence is *may-testing* equivalence [13]. From this perspective,  $-$  and  $-\boxed{x}-\boxed{x}-$  are not *must equivalent*, since the former must pass the test  $\bullet-; -; -\bullet$  while  $-\boxed{x}-\boxed{x}-$  may not. It is worth to remark here that the distinction between may and must testing will cease to make sense in Section 5 where we identify a certain class of circuits equipped with a proper flow directionality and thus a deterministic, input-output, behaviour.

**Theorem 2 (Full abstraction).**  $c \equiv d$  iff  $c \stackrel{\text{allH}}{=} d$

The remainder of this section is devoted to the proof of Theorem 2. We will start by clarifying the relationship between fractions of polynomials (the denotational domain) and trajectories (the operational domain).

### 4.1 From Polynomial Fractions to Trajectories

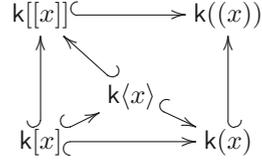
The missing link between polynomial fractions and trajectories are (*formal*) *Laurent series*: we now recall this notion. Formally, a Laurent series is a function  $\sigma: \mathbb{Z} \rightarrow \mathbb{k}$  for which there exists  $j \in \mathbb{Z}$  such that  $\sigma(i) = 0$  for all  $i < j$ . We write  $\sigma$  as  $\dots, \sigma(-1), \underline{\sigma(0)}, \sigma(1), \dots$  with position 0 underlined, or as formal sum  $\sum_{i=d}^{\infty} \sigma(i)x^i$ . Each Laurent series  $\sigma$  has then a *degree*  $d \in \mathbb{Z}$ , which is the first non-zero element. Laurent series form a field  $\mathbb{k}((x))$ : sum is pointwise, product is by convolution, and the inverse  $\sigma^{-1}$  of  $\sigma$  with degree  $d$  is defined as:

$$\sigma^{-1}(i) = \begin{cases} 0 & \text{if } i < -d \\ \sigma(d)^{-1} & \text{if } i = -d \\ \frac{\sum_{i=1}^n (\sigma(d+i) \cdot \sigma^{-1}(-d+n-i))}{-\sigma(d)} & \text{if } i = -d+n \text{ for } n > 0 \end{cases} \quad (9)$$

Note (formal) power series, which form ‘just’ a ring  $\mathbb{k}[[x]]$ , are a particular case of Laurent series, namely those  $\sigma$ s for which  $d \geq 0$ . What is most interesting for our purposes is how polynomials and fractions of polynomials relate to  $\mathbb{k}((x))$  and  $\mathbb{k}[[x]]$ . First, the ring  $\mathbb{k}[x]$  of polynomials embeds into  $\mathbb{k}[[x]]$ , and thus into  $\mathbb{k}((x))$ : a polynomial  $p_0 + p_1x + \dots + p_nx^n$  can also be regarded as the power series  $\sum_{i=0}^{\infty} p_i x^i$  with  $p_i = 0$  for all  $i > n$ . Because Laurent series are closed under division, this immediately gives also an embedding of the field of polynomial fractions  $\mathbb{k}(x)$  into  $\mathbb{k}((x))$ . Note that the full expressiveness of  $\mathbb{k}((x))$  is required: for instance, the fraction  $\frac{1}{x}$  is represented as the Laurent series  $\dots, 0, 1, \underline{0}, 0, \dots$ ,

which is not a power series, because a non-zero value appears before position 0. In fact, fractions that are expressible as power series are precisely the *rational* fractions, i.e. of the form  $\frac{k_0+k_1x+k_2x^2+\dots+k_nx^n}{l_0+l_1x+l_2x^2+\dots+l_nx^n}$  where  $l_0 \neq 0$ .

Rational fractions form a ring  $k\langle x \rangle$  which, differently from the full field  $k(x)$ , embeds into  $k[[x]]$ . Indeed, whenever  $l_0 \neq 0$ , the inverse of  $l_0 + l_1x + l_2x^2 \dots + l_nx^n$  is, by (9), a *bona fide* power series. The commutative diagram on the right is a summary.



Relations between  $k((x))$ -vectors organise themselves into a prop  $\mathbf{Arel}_{k((x))}$  (see Definition 2). There is an evident prop morphism  $\iota: \mathbf{Arel}_{k(x)} \rightarrow \mathbf{Arel}_{k((x))}$ : it maps the empty affine relation on  $k(x)$  to the one on  $k((x))$ , and otherwise applies pointwise the embedding of  $k(x)$  into  $k((x))$ . For the next step, observe that trajectories are in fact rearrangements of Laurent series: each pair of vectors  $(u, v) \in k((x))^n \times k((x))^m$ , as on the left below, yields the trajectory  $\kappa(u, v)$  defined for all  $i \in \mathbb{Z}$  as on the right below.

$$(u, v) = \left( \begin{pmatrix} \alpha^1 \\ \vdots \\ \alpha^n \end{pmatrix}, \begin{pmatrix} \beta^1 \\ \vdots \\ \beta^m \end{pmatrix} \right) \quad \kappa(u, v)(i) = \left( \begin{pmatrix} \alpha^1(i) \\ \vdots \\ \alpha^n(i) \end{pmatrix}, \begin{pmatrix} \beta^1(i) \\ \vdots \\ \beta^m(i) \end{pmatrix} \right)$$

Similarly to  $\iota$ , the assignment  $\kappa$  extends to sets of vectors, and also to a prop morphism from  $\mathbf{Arel}_{k((x))}$  to  $\mathbf{Traj}$ . Together,  $\kappa$  and  $\iota$  provide the desired link between operational and denotational semantics.

**Theorem 3.**  $\langle \cdot \rangle = \kappa \circ \iota \circ \llbracket \cdot \rrbracket$

*Proof.* Since both are symmetric monoidal functors from a free prop, it is enough to check the statement for the generators of  $\mathbf{ACirc}$ . We show, as an example, the case of  $\text{---}\bullet\text{---}$ . By Definition 3,  $\llbracket \text{---}\bullet\text{---} \rrbracket = \left\{ \left( p, \begin{pmatrix} p \\ p \end{pmatrix} \right) \mid p \in k(x) \right\}$ . This is mapped by  $\iota$  to  $\left\{ \left( \alpha, \begin{pmatrix} \alpha \\ \alpha \end{pmatrix} \right) \mid \alpha \in k((x)) \right\}$ . Now, to see that  $\kappa(\iota(\llbracket \text{---}\bullet\text{---} \rrbracket)) = \langle \text{---}\bullet\text{---} \rangle$ , it is enough to observe that a trajectory  $\sigma$  is in  $\kappa(\iota(\llbracket \text{---}\bullet\text{---} \rrbracket))$  precisely when, for all  $i$ , there exists some  $k_i \in k$  such that  $\sigma(i) = \left( k_i, \begin{pmatrix} k_i \\ k_i \end{pmatrix} \right)$ .  $\square$

### 4.2 Proof of Full Abstraction

We now have the ingredients to prove Theorem 2. First, we prove an adequacy result for  $(0, 0)$  circuits.

**Proposition 5.** *Let  $c \in \mathbf{ACirc}[0, 0]$ . Then  $\llbracket c \rrbracket = id_0$  if and only if  $c \uparrow$ .*

*Proof.* By Proposition 3, either  $\llbracket c \rrbracket = id_0$  or  $\llbracket c \rrbracket = \emptyset$ , which, combined with Theorem 3, means that  $\langle c \rangle = \kappa \circ \iota(id_0)$  or  $\langle c \rangle = \kappa \circ \iota(\emptyset)$ . By definition of  $\iota$  this implies that either  $\langle c \rangle$  contains a trajectory or not. In the first case  $c \uparrow$ ; in the second  $c \not\uparrow$ .  $\square$

Next we obtain a result that relates denotational equality in all contexts to equality in  $\mathfrak{C}\mathfrak{H}$ . Note that it is not trivial: since we consider ground contexts it does not make sense to merely consider “identity” contexts. Instead, it is at this point that we make another crucial use of affinity, taking advantage of the increased expressivity of affine circuits, as showcased by Proposition 2.

**Proposition 6.** *If  $\llbracket C[c] \rrbracket = \llbracket C[d] \rrbracket$  for all contexts  $C[-]$ , then  $c \stackrel{\mathfrak{C}\mathfrak{H}}{=} d$ .*

*Proof.* Suppose that  $c \not\stackrel{\mathfrak{C}\mathfrak{H}}{=} d$ . Then  $\llbracket c \rrbracket \neq \llbracket d \rrbracket$ . Since both  $\llbracket c \rrbracket$  and  $\llbracket d \rrbracket$  are affine relations over  $k(x)$ , there exists a pair of vectors  $(u, v) \in k(x)^n \times k(x)^m$  that is in one of  $\llbracket c \rrbracket$  and  $\llbracket d \rrbracket$ , but not both. Assume wlog that  $(u, v) \in \llbracket c \rrbracket$  and  $(u, v) \notin \llbracket d \rrbracket$ . By Proposition 2, there exists  $c_u$  and  $c_v$  such that  $\llbracket c_u ; c ; c_v \rrbracket = \llbracket c_u \rrbracket ; \llbracket c \rrbracket ; \llbracket c_v \rrbracket = \{(\bullet, u)\} ; \llbracket c \rrbracket ; \{(v, \bullet)\}$ . Since  $(u, v) \in \llbracket c \rrbracket$ , then  $\llbracket c_u ; c ; c_v \rrbracket = \{(\bullet, \bullet)\}$ . Instead, since  $(u, v) \notin \llbracket d \rrbracket$ , we have that  $\llbracket c_u ; d ; c_v \rrbracket = \emptyset$ . Therefore, for the context  $C[-] = c_u ; - ; c_v$ , we have that  $\llbracket C[c] \rrbracket \neq \llbracket C[d] \rrbracket$ .  $\square$

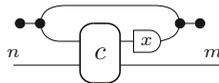
The proof of our main result is now straightforward.

*Proof of Theorem 2.* Let us first suppose that  $c \stackrel{\mathfrak{C}\mathfrak{H}}{=} d$ . Then  $\llbracket C[c] \rrbracket = \llbracket C[d] \rrbracket$  for all contexts  $C[-]$ , since  $\llbracket \cdot \rrbracket$  is a morphism of props. By Corollary 5, it follows immediately that  $C[c] \uparrow$  if and only if  $C[d] \uparrow$ , namely  $c \equiv d$ .

Conversely, suppose that, for all  $C[-]$ ,  $C[c] \uparrow$  iff  $C[d] \uparrow$ . Again by Corollary 5, we have that  $\llbracket C[c] \rrbracket = \llbracket C[d] \rrbracket$ . We conclude by invoking Proposition 6.  $\square$

## 5 Functional Behaviour and Signal Flow Graphs

There is a sub-prop **SF** of **Circ** of classical *signal flow graphs* (see e.g. [21]). Here signal flows left-to-right, possibly featuring *feedback loops*, provided that these go through at least one register. Feedback can be captured algebraically via an operation  $\text{Tr}(\cdot) : \text{Circ}[n + 1, m + 1] \rightarrow \text{Circ}[n, m]$  taking  $c : n + 1 \rightarrow m + 1$  to:



Following [9], let us call  $\text{Circ}^{\rightarrow}$  the free sub-prop of **Circ** of circuits built from (3) and the generators of (1), without  $\vdash$ . Then **SF** is defined as the closure of  $\text{Circ}^{\rightarrow}$  under  $\text{Tr}(\cdot)$ . For instance, the circuit of Example 2 is in **SF**.

Signal flow graphs are intimately connected to the executability of circuits. In general, the rules of Figure 2 do not assume a fixed flow orientation. As a result, some circuits in **Circ** are not executable as *functional input-output* systems, as we have demonstrated with  $\vdash \overline{x} \vdash$ ,  $\vdash \circ$  and  $\overline{x} \overline{x}$  of Examples 3-5. Notice that none of these are signal flow graphs. In fact, the circuits of **SF** do not have pathological behaviour, as we shall state more precisely in Proposition 9.

At the denotational level, signal flow graphs correspond precisely to *rational* functional behaviours, that is, matrices whose coefficients are in the ring  $k(x)$

of *rational fractions* (see Section 4.1). We call such matrices, rational matrices. One may check that the semantics of a signal flow graph  $c: (n, m)$  is always of the form  $\llbracket c \rrbracket = \{(v, A \cdot v) \mid v \in \mathbf{k}(x)^n\}$ , for some  $m \times n$  rational matrix  $A$ . Conversely, all relations that are the graph of rational matrices can be expressed as signal flow graphs.

**Proposition 7.** *Given  $c: (n, m)$ , we have  $\llbracket c \rrbracket = \{(p, A \cdot p) \mid p \in \mathbf{k}(x)^n\}$  for some rational  $m \times n$  matrix  $A$  iff there exists a signal flow graph  $f$ , i.e., a circuit  $f: (n, m)$  of SF, such that  $\llbracket f \rrbracket = \llbracket c \rrbracket$ .*

*Proof.* This is a folklore result in control theory which can be found in [30]. The details of the translation between rational matrices and circuits of SF can be found in [10, Section 7].  $\square$

The following gives an alternative characterisation of rational matrices—and therefore, by Proposition 7, of the behaviour of signal flow graphs—that clarifies their role as realisations of circuits.

**Proposition 8.** *An  $m \times n$  matrix is rational iff  $A \cdot r \in \mathbf{k}\langle x \rangle^m$  for all  $r \in \mathbf{k}\langle x \rangle^n$ .*

Proposition 8 is another guarantee of good behaviour—it justifies the name of inputs (resp. outputs) for the left (resp. right) ports of signal flow graphs. Recall from Section 4.1 that rational fractions can be mapped to Laurent series of nonnegative degree, i.e., to plain power series. Operationally, these correspond to trajectories that start after  $t = 0$ . Proposition 8 guarantees that any trajectory of a signal flow graph whose first nonzero value on the left appears at time  $t = 0$ , will not have nonzero values on the right starting before time  $t = 0$ . In other words, signal flow graphs can be seen as processing a stream of values from left to right. As a result, their ports can be clearly partitioned into inputs and outputs.

But the circuits of SF are too restrictive for our purposes. For example,  can also be seen to realise a functional behaviour transforming inputs on the left into outputs on the right yet it is not in SF. Its behaviour is no longer linear, but affine. Hence, we need to extend signal flow graphs to include functional affine behaviour. The following definition does just that.

**Definition 10.** *Let ASF be the sub-prop of ACirc obtained from all the generators in (1), closed under  $\text{Tr}(\cdot)$ . Its circuits are called affine signal flow graphs.*

As before, none of  $\vdash \boxed{x} \dashv$ ,  $\vdash \circ$  and  $\dashv \boxed{x} \dashv \boxed{x}$  from Examples 3-5 are affine signal flow graphs. In fact, ASF rules out pathological behaviour: all computations can be extended to be infinite, or in other words, do not get stuck.

**Proposition 9.** *Given an affine signal flow graph  $f$ , for every computation*

$$t \triangleright f_0 \xrightarrow[u_t]{u_t} f_1 \xrightarrow[v_{p+1}]{u_{t+1}} \dots f_n$$

*there exists a trajectory  $\sigma \in \langle c \rangle$  such that  $\sigma(i) = (u_i, v_i)$  for  $t \leq i \leq t + n$ .*

*Proof.* By induction on the structure of affine signal flow graphs.  $\square$

If SF circuits correspond precisely to  $k\langle x \rangle$ -matrices, those of ASF correspond precisely to  $k\langle x \rangle$ -affine transformations.

**Definition 11.** A map  $f: k(x)^n \rightarrow k(x)^m$  is an affine map if there exists an  $m \times n$  matrix  $A$  and  $b \in k(x)^m$  such that  $f(p) = A \cdot p + b$  for all  $p \in k(x)^n$ . We call the pair  $(A, b)$  the representation of  $f$ .

The notion of rational affine map is a straightforward extension of the linear case and so is the characterisation in terms of rational input-output behaviour.

**Definition 12.** An affine map  $f: p \mapsto A \cdot p + b$  is rational if  $A$  and  $b$  have coefficients in  $k\langle x \rangle$ .

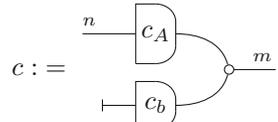
**Proposition 10.** An affine map  $f: k(x)^n \rightarrow k(x)^m$  is rational iff  $f(r) \in k\langle x \rangle^m$  for all  $r \in k(x)^n$ .

The following extends the correspondence of Proposition 7, showing that ASF is the rightful affine heir of SF.

**Proposition 11.** Given  $c: (n, m)$ , we have  $\llbracket c \rrbracket = \{(p, f(p)) \mid p \in k(x)^n\}$  for some rational affine map  $f$  iff there exists an affine signal flow graph  $g$ , i.e., a circuit  $g: (n, m)$  of ASF, such that  $\llbracket g \rrbracket = \llbracket c \rrbracket$ .

*Proof.* Let  $f$  be given by  $p \mapsto Ap + b$  for some rational  $m \times n$  matrix  $A$  and vector  $b \in k\langle x \rangle^m$ . By Proposition 7, we can find a circuit  $c_A$  of SF such that

$\llbracket c_A \rrbracket = \{(p, A \cdot p) \mid p \in k(x)\}$ . Similarly, we can represent  $b$  as a signal flow graph  $c_b$  of sort  $(1, m)$ . Then, the circuit on the right is clearly in ASF and verifies  $\llbracket c \rrbracket = \{(p, Ap + b) \mid p \in k(x)\}$  as required.



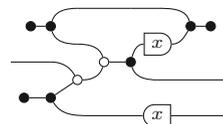
For the converse direction it is straightforward to check by structural induction that the denotation of affine signal flow graphs is the graph (in the set-theoretic sense of pairs of values) of some rational affine map.  $\square$

## 6 Realisability

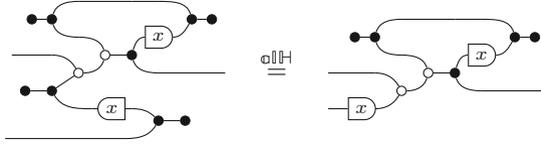
In the previous section we gave a restricted class of morphisms with good behavioural properties. We may wonder how much of ACirc we can capture with this restricted class. The answer is, in a precise sense: most of it.

Surprisingly, the behaviours realisable in Circ—the purely linear fragment—are not more expressive. In fact, from an operational (or denotational, by full abstraction) point of view, Circ is nothing more than jumbled up version of SF. Indeed, it turns out that Circ enjoys a *realisability* theorem: any circuit  $c$  of Circ can be associated with one of SF, that implements or realises the behaviour of  $c$  into an executable form.

But the corresponding realisation may not flow neatly from left to right like signal flow graphs do—its inputs and outputs may have been moved from one side to the other. Consider for example, the circuit on the right



It does not belong to SF but it can be read as a signal flow graph with an input that has been bent and moved to the bottom right. The behaviour it realises can therefore be executed by rewiring this port to obtain a signal flow graph:



We will not make this notion of rewiring precise here but refer the reader to [9] for the details. The intuition is simply that a rewiring partitions the ports of a circuit into two sets—that we call inputs and outputs—and uses  $\bullet\curvearrowright$  or  $\curvearrowleft\bullet$  to bend input ports to the left and output ports to the right. The realisability theorem then states that we can always recover a (not necessarily unique) signal flow graph from any circuit by performing these operations.

**Theorem 4.** [9, Theorem 5] *Every circuit in Circ is equivalent to the rewiring of a signal flow graph, called its realisation.*

This theorem allows us to extend the notion of inputs and outputs to all circuits of Circ.

**Definition 13.** *A port of a circuit  $c$  of Circ is an input (resp. output) port, if there exists a realisation for which it is an input (resp. output).*

Note that, since realisations are not necessarily unique, the same port can be both an input and an output. Then, the realisability theorem (Theorem 4) says that every port is always an input, an output or both (but never neither).

An output-only port is an output port that is not an input port. Similarly an input-only port is an input port that is not an output port.

*Example 8.* The left port of the register  $\boxed{x}$  is input-only whereas its right port is output-only. In the identity wire, both ports are input and output ports. The single port of  $\circ$  is output-only ; that of  $\bullet$  is input-only.

While in the purely linear case, all behaviours are realisable, the general case of ACirc is a bit more subtle. To make this precise, we can extend our definition of realisability to include affine signal flow graphs.

**Definition 14.** *A circuit of ACirc is realisable if its ports can be rewired so that it is equivalent to a circuit of ASF.*

*Example 9.*  $\dashv$  is realisable;  $\dashv\boxed{x}$  is not.

Notice that Proposition 11, gives the following equivalent semantic criterion for realisability. Realisable behaviours are precisely those that map rationals to rationals.

**Theorem 5.** *A circuit  $c$  is realisable iff its ports can be partitioned into two sets, that we call inputs and outputs, such that the corresponding rewiring of  $c$  is an affine rational map from inputs to outputs.*

We offer another perspective on realisability below: realisable behaviours correspond precisely to those for which the  $\vdash$  constants are connected to inputs of the underlying *Circ*-circuit. First, notice that, since

$$\vdash \bullet \left( \begin{array}{c} \text{---} \\ \text{---} \end{array} \right) \stackrel{(1\text{-dup})}{=} \begin{array}{c} \text{---} \\ \text{---} \end{array} \quad \text{and} \quad \vdash \bullet \stackrel{(1\text{-del})}{=} \square$$

in  $\mathcal{C}\mathbb{H}$ , we can assume without loss of generality that each circuit contains exactly one  $\vdash$ .

**Proposition 12.** *Every circuit  $c$  of  $\text{ACirc}$  is equivalent to one with precisely one  $\vdash$  and no  $\dashv$ .*

For  $c: (n, m)$  a circuit of  $\text{ACirc}$ , we will call  $\hat{c}$  the circuit of *Circ* of sort  $(n + 1, m)$  that one obtains by first transforming  $c$  into an equivalent circuit with a single  $\vdash$  and no  $\dashv$  as above, then removing this  $\vdash$ , and replacing it by an identity wire that extends to the left boundary.

**Theorem 6.** *A circuit  $c$  is realisable iff  $\vdash$  is connected to an input port of  $\hat{c}$ .*

## 7 Conclusion and Future Work

We introduced the operational semantics of the *affine* extension of the signal flow calculus and proved that contextual equivalence coincides with denotational equality, previously introduced and axiomatised in [6]. We have observed that, at the denotational level, affinity provides two key properties (Propositions 2 and 3) for the proof of full abstraction. However, at the operational level, affinity forces us to consider computations starting in the *past* (Example 3) as the syntax allows terms lacking a proper flow directionality. This leads to circuits that might deadlock ( $\dashv$  in Example 4) or perform some problematic computations ( $\dashv \square \dashv$  in Example 5). We have identified a proper subclass of circuits, called affine signal flow graphs (Definition 10), that possess an inherent flow directionality: in these circuits, the same pathological behaviours do not arise (Proposition 9). This class is not too restrictive as it captures all desirable behaviours: a realisability result (Theorem 5) states that all and only the circuits that do not need computations to start in the past are equivalent to (the rewiring of) an affine signal flow graph.

The reader may be wondering why we do not restrict the syntax to affine signal flow graphs. The reason is that, like in the behavioural approach to control theory [33], the lack of flow direction is what allows the (affine) signal flow calculus to achieve a strong form of compositionality and a complete axiomatisation (see [9] for a deeper discussion).

We expect that similar methods and results can be extended to other models of computation. Our next step is to tackle Petri nets, which, as shown in [5], can be regarded as terms of the signal flow calculus, but over  $\mathbb{N}$  rather than a field.

## References

1. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS), 2004. pp. 415–425. IEEE (2004)
2. Baez, J., Erbele, J.: Categories in control. *Theory and Applications of Categories* **30**, 836–881 (2015)
3. Baez, J.C.: Network theory (2014), <http://math.ucr.edu/home/baez/networks/>, website (retrieved 15/04/2014)
4. Basold, H., Bonsangue, M., Hansen, H., Rutten, J.: (Co)Algebraic characterizations of signal flow graphs. In: van Breugel, F., Kashefi, E., Palamidessi, C., Rutten, J. (eds.) *Horizons of the Mind. A Tribute to Prakash Panangaden*, Lecture Notes in Computer Science, vol. 8464, pp. 124–145. Springer International Publishing (2014)
5. Bonchi, F., Holland, J., Piedeleu, R., Sobociński, P., Zanasi, F.: Diagrammatic algebra: from linear to concurrent systems. *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* **3**, 1–28 (2019)
6. Bonchi, F., Piedeleu, R., Sobociński, P., Zanasi, F.: Graphical affine algebra. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–12 (2019)
7. Bonchi, F., Piedeleu, R., Sobociński, P., Zanasi, F.: Contextual equivalence for signal flow graphs (2020), <https://arxiv.org/abs/2002.08874>
8. Bonchi, F., Sobociński, P., Zanasi, F.: A categorical semantics of signal flow graphs. In: *Proceedings of the 25th International Conference on Concurrency Theory (CONCUR)*. pp. 435–450. Springer (2014)
9. Bonchi, F., Sobocinski, P., Zanasi, F.: Full abstraction for signal flow graphs. In: *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. pp. 515–526 (2015)
10. Bonchi, F., Sobocinski, P., Zanasi, F.: The calculus of signal flow diagrams I: linear relations on streams. *Information and Computation* **252**, 2–29 (2017)
11. Coecke, B., Duncan, R.: Interacting quantum observables. In: *Proceedings of the 35th international colloquium on Automata, Languages and Programming (ICALP), Part II*. pp. 298–310 (2008)
12. Coecke, B., Kissinger, A.: *Picturing Quantum Processes - A first course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press (2017)
13. De Nicola, R., Hennessy, M.C.: Testing equivalences for processes. *Theoretical Computer Science* **34**(1-2), 83–133 (1984)
14. Ghica, D.R.: Diagrammatic reasoning for delay-insensitive asynchronous circuits. In: *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pp. 52–68. Springer (2013)
15. Ghica, D.R., Jung, A.: Categorical semantics of digital circuits. In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD)*. pp. 41–48 (2016)
16. Ghica, D.R., Lopez, A.: A structural and nominal syntax for diagrams. In: *Proceedings 14th International Conference on Quantum Physics and Logic (QPL)*. pp. 71–83 (2017)
17. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall (1985)
18. Honda, K., Yoshida, N.: On reduction-based process semantics. *Theoretical Computer Science* **152**(2), 437–486 (1995)

19. Mac Lane, S.: Categorical algebra. *Bulletin of the American Mathematical Society* **71**, 40–106 (1965)
20. Mac Lane, S.: *Categories for the Working Mathematician*. Springer (1998)
21. Mason, S.J.: *Feedback Theory: I. Some Properties of Signal Flow Graphs*. MIT Research Laboratory of Electronics (1953)
22. Milius, S.: A sound and complete calculus for finite stream circuits. In: *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science (LICS)*. pp. 421–430 (2010)
23. Milner, R.: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92. Springer (1980)
24. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP)*. pp. 685–695 (1992)
25. Morris Jr, J.H.: *Lambda-calculus models of programming languages*. Ph.D. thesis, Massachusetts Institute of Technology (1969)
26. Pavlovic, D.: Monoidal computer I: Basic computability by string diagrams. *Information and Computation* **226**, 94–116 (2013)
27. Pavlovic, D.: Monoidal computer II: Normal complexity by string diagrams. arXiv:1402.5687 (2014)
28. Plotkin, G.D.: Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science* **1**(2), 125–159 (1975)
29. Rutten, J.J.M.M.: A tutorial on coinductive stream calculus and signal flow graphs. *Theoretical Computer Science* **343**(3), 443–481 (2005)
30. Rutten, J.J.M.M.: Rational streams coalgebraically. *Logical Methods in Computer Science* **4**(3) (2008)
31. Selinger, P.: A survey of graphical languages for monoidal categories. *Springer Lecture Notes in Physics* **13**(813), 289–355 (2011)
32. Shannon, C.E.: *The theory and design of linear differential equation machines*. Tech. rep., National Defence Research Council (1942)
33. Willems, J.C.: The behavioural approach to open and interconnected systems. *IEEE Control Systems Magazine* **27**, 46–99 (2007)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

