Simple Modal Types for Functional Reactive Programming

PATRICK BAHR, IT University of Copenhagen, Denmark

Functional reactive programming (FRP) is a declarative programming paradigm for implementing reactive programs at a high level of abstraction. It applies functional programming principles to construct and manipulate time-varying values, also known as *signals*. However, for this programming paradigm to work in practice, an FRP language must ensure that programs are *causal*, *productive*, and free of *space leaks*. Over the past fifteen years, several modal type systems to enforce these operational properties have been developed.

We present a new FRP language with a significantly simplified modal type system that imposes fewer restrictions than previous modal FRP languages while still guaranteeing the central operational properties of causality, productivity, and absence of space leaks. The key enabling idea is to define the language's semantics so that it is impossible to inspect a signal's past. In particular, a signal only stores its current value and how to compute its future values, but it cannot retain its past values. As a result, we obtain a language with a simpler modal type system that is also more expressive and supports a more modular programming style compared to previous modal FRP calculi without space leaks. While the central idea applies to both synchronous and asynchronous FRP, we focus here on the more challenging asynchronous case.

1 Introduction

Functional reactive programming (FRP) [Elliott and Hudak 1997; Wan and Hudak 2000] is a declarative programming paradigm for implementing reactive programs. In turn, a reactive program is an indefinitely running piece of software that interacts with its environment by receiving input from the environment and, in return, producing output that is sent back to the environment. This class of software is ubiquitous – ranging from graphical user interfaces and servers to safety-critical control software for components in aircraft and power plants.

The central abstraction employed by FRP to model the interaction with the environment is the notion of a *signal* [Courtney and Elliott 2001] (or *behaviour* [Elliott and Hudak 1997]), which represents a time-varying value. FRP uses functional programming principles to construct, consume, and manipulate such signals. This approach affords a high level of abstraction and modularity, along with equational reasoning principles.

However, this high level of abstraction makes it difficult to compile FRP programs into effective and efficient low-level software. To allow such low-level implementations in general, the FRP language must guarantee three properties: *productivity*, that is, after receiving an input, the program must produce its response in finite time; *causality*, that is, at any time, the output may depend only on current or past inputs; and *no space leaks*, that is, the program may not retain input values in memory indefinitely. Reactive programs that do not satisfy these properties will eventually grind to a halt and thus become unresponsive.

Several approaches have been proposed to guarantee some or all of these operational properties (cf. section 6). In this article, we are particularly interested in a type-based approach, which allows programmers direct access to signals but carefully controls that access using modal types inspired by linear temporal logic [Pnueli 1977]. Originally proposed by Krishnaswami and Benton [2011b], this approach has spawned several FRP calculi that use modal types to guarantee productivity, causality, and the absence of space leaks [Bahr 2022; Bahr et al. 2019; Krishnaswami 2013]. The type systems of these calculi feature two type modalities: the *later* modality $\bigcirc$ to classify data that is available in the next time step and the *stable* modality $\Box$ to classify data that can be kept in memory across time steps without causing space leaks.

Bahr and Møgelberg [2023] have shown that this approach extends to *asynchronous* FRP, where there is no global clock and where signals thus may update asynchronously from one another. Asynchronous FRP provides a more suitable programming model for systems that lack a uniform

global clock such as GUIs [Graulund et al. 2021]. To capture this asynchronicity with a modal type system, Bahr and Møgelberg's Async RaTT calculus replaces the synchronous later modality $\bigcirc$ with an asynchronous later modality $\textcircled{\exists}$. While a value of type $\bigcirc A$ is a delayed computation that produces a value of type A at the next tick of the global clock, a value of type $\textcircled{\exists}\, A$ is a pair (θ, f) consisting of a clock θ and a delayed computation f that produces a value of type A as soon as θ ticks. *Asynchronous* signals are then represented by the guarded recursive type $\mathrm{Sig}\ A \cong A \times \textcircled{\exists}\ (\mathrm{Sig}\ A)$. That is, a value of type $\mathrm{Sig}\ A$ is of the form $(v, (\theta, f))$ consisting of the signal's current value $v : A$, a clock θ, and a delayed computation f that produces a new value of type $\mathrm{Sig}\ A$ as soon as θ ticks. Consequently, we can intuitively think of an asynchronous signal in Async RaTT as a (possibly infinite) sequence $v_0 \langle\theta_0\rangle\, v_1 \langle\theta_1\rangle \ldots$ consisting of lazily computed clocks θ_i and values v_i of type A, each of which only becomes available after the preceding clocks $\theta_0, \ldots, \theta_{i-1}$ have ticked *in succession*. For example, θ_2 and v_2 are available after first θ_0 and then θ_1 have ticked.

In this article, we present a new asynchronous FRP language, called Rizzo, that uses the same representation of asynchronous signals as Async RaTT [Bahr and Møgelberg 2023], i.e. $\mathrm{Sig}\ A \cong A \times \textcircled{\exists}\ (\mathrm{Sig}\ A)$. However, instead of using a sophisticated Fitch-style type system [Clouston 2018] and an additional type modality $\Box$ to rule out space leaks, we define the semantics of Rizzo to rule out space leaks *by construction*. This allows us to give Rizzo a much simpler type system and to extend its expressiveness compared to Async RaTT – all while maintaining the three crucial operational properties. The key idea is that when evaluating a signal $s : \mathrm{Sig}\ A$ in response to a sequence of events τ, we only keep the current value of the signal and discard all its history. That is, given a signal s that denotes a possibly infinite sequence $v_0 \langle\theta_0\rangle\, v_1 \langle\theta_1\rangle \ldots$ and a finite sequence of events τ that causes the clocks $\theta_0, \ldots, \theta_{i-1}$ to tick (but not θ_i), the evaluation of s in response to τ, written $s \rightsquigarrow_\tau v$, produces a signal $v : \mathrm{Sig}\ A$ that denotes the sequence $v_i \langle\theta_i\rangle\, v_{i+1} \langle\theta_{i+1}\rangle \ldots$, i.e. forgetting all its prior history from before θ_{i-1} ticked.

This forgetful semantics applies to the language as a whole: A term $t : A$ evaluates to a value $v : A$ in response to τ, written $t \rightsquigarrow_\tau v$. For example, we may have a GUI program $t : \mathit{Widget}$ that implements the tree structure describing the GUI (such as the DOM of a website), so that if $t \rightsquigarrow_\tau v$, then the value v is the state of the GUI after it has received the events τ. Importantly, we give Rizzo an operational semantics that will compute $t \rightsquigarrow_\tau v$ *incrementally*. That is, after we computed $t \rightsquigarrow_\tau v$ and then received an additional event e, we can compute $t \rightsquigarrow_{\tau,e} v'$ from v and e alone. Moreover, this update of v to v' is performed *in place*, only replacing the parts of v that have changed.

The main technical results of this article are the metatheoretical properties of Rizzo: We prove that the language is productive, causal, and free of space leaks. The proof uses a combination of a logical relations argument and a type preservation argument. In addition, this article presents multiple examples that demonstrate the expressiveness and the simplicity of Rizzo compared to previous work. The proofs of the metatheoretical results and all Rizzo examples in this article have been fully formalised using the *Lean* theorem prover [Moura and Ullrich 2021]. This Lean formalisation is available in the supplementary material [Bahr 2026].

Overview. We introduce Rizzo in section 2 and demonstrate its expressiveness in section 3. Section 4 presents the operational semantics of Rizzo and gives a precise account of the main technical results. Section 5 sketches the proof of the main results. Finally, section 6 discusses related work, and section 7 discusses conclusions and future work.

2 Introduction to Rizzo

We give an overview of the Rizzo language, its type system, and an informal account of its semantics. For a complete specification of its syntax and type system, we refer the reader to Figs. 1, 2, and 3. The formal semantics is presented in detail in section 4.

$$
\begin{array}{lll}
\text{Channels} & \kappa \in \mathsf{Chan}, & \text{Locations } l \in \mathsf{Loc} \\
\text{Events} & e & ::= \kappa \mapsto t \\
\text{Types} & A, B, F, G & ::= \alpha \mid 1 \mid F \times G \mid F + G \mid F \to G \mid \triangleq F \mid \triangledown F \mid \mathsf{Sig}\, F \mid \mathsf{Chan}\, F \mid \mu\alpha.F \\
\text{Values} & v, w & ::= () \mid \lambda x.t \mid (v, w) \mid \mathsf{in}_i\, v \mid v ::_A w \mid \kappa \mid \mathsf{wait}\, \kappa \mid \mathsf{watch}\, v \mid v \mathbin{\triangleright\!\!=} w \mid \mathsf{delay}\, t \\
& & \mid \ \mathsf{never} \mid \mathsf{select}\, v\, w \mid \mathsf{cons}_{\mu\alpha.F}\, v \\
\text{Terms} & s, t & ::= v \mid x \mid \mathsf{rec}(x.s, t) \mid (s, t) \mid \mathsf{in}_i\, t \mid \pi_i\, t \mid t_1 t_2 \mid \mathsf{chan}_A \mid \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 \mid l \\
& & \mid \ s ::_A t \mid \mathsf{select}\, s\, t \mid s \mathbin{\triangleright\!\!=} t \mid s \circledast t \mid \mathsf{wait}\, t \mid \mathsf{watch}\, t \mid \mathsf{fix}\, x.t \mid \mathsf{head}\, t \mid \mathsf{tail}\, t \mid \mathsf{cons}_{\mu\alpha.F}\, t
\end{array}
$$

Fig. 1. Syntax.

$$
\frac{\Phi \vdash F : \mathsf{type}}{\Phi \vdash \mathsf{Sig}\, F : \mathsf{type}} \qquad
\frac{\Phi \vdash F : \mathsf{type} \quad \Phi \vdash G : \mathsf{type}}{\Phi \vdash F \times G : \mathsf{type}} \qquad
\frac{\Phi \vdash F : \mathsf{type} \quad \Phi \vdash G : \mathsf{type}}{\Phi \vdash F + G : \mathsf{type}} \qquad
\frac{\vdash F : \mathsf{type} \quad \Phi \vdash G : \mathsf{type}}{\Phi \vdash F \to G : \mathsf{type}}
$$

$$
\frac{}{\Phi \vdash 1 : \mathsf{type}} \qquad
\frac{\vdash F : \mathsf{type}}{\Phi \vdash \triangleq F : \mathsf{type}} \qquad
\frac{\vdash F : \mathsf{type}}{\Phi \vdash \triangledown F : \mathsf{type}} \qquad
\frac{\vdash F : \mathsf{type}}{\Phi \vdash \mathsf{Chan}\, F : \mathsf{type}} \qquad
\frac{\Phi, \alpha \vdash F : \mathsf{type}}{\Phi \vdash \mu\, \alpha.F : \mathsf{type}} \qquad
\frac{\alpha \in \Phi}{\Phi \vdash \alpha : \mathsf{type}}
$$

Fig. 2. Type formation rules.

The language is based on the simply typed lambda calculus with product, sum, and inductive types. To account for well-formed inductive types of the form $\mu\alpha.F$, we have an explicit type formation judgement $\Phi \vdash F : \mathsf{type}$ defined in Fig. 2. It states that F is a well-formed type with free type variables drawn from Φ, and we call F a *closed type* if $\vdash F : \mathsf{type}$. By convention, we use A, B, C, and D to range over closed types, while F and G range over possibly open types. We look more closely at the type formation rules in section 2.3 when we discuss recursive types. The typing judgement $\Gamma \vdash_\Delta t : A$ defined in Fig. 3 takes a term t, a closed type A, a *typing context* Γ consisting of variable type assignments of the form $x : B$, and a *channel context* Δ consisting of channel type assignments of the form $\kappa : \mathsf{Chan}\, B$. We discuss channels in more detail in section 2.2 below. The term language defined in Fig. 1 also includes locations $l \in \mathsf{Loc}$, but these are not typable by the typing judgement and are only relevant for the operational semantics discussed in section 4.

2.1 Signals and Delayed Computations

Similarly to Bahr and Møgelberg [2023], Rizzo features two later modalities to capture asynchronicity: an *existential* later modality $\triangleq$ and a *universal* later modality $\triangledown$. Intuitively speaking, a value of type $\triangleq A$ is a pair (θ, f) consisting of a *clock* θ and a delayed computation f that will produce a value of type A as soon as θ *ticks*. Given a value $v : \triangleq A$, we write $\mathsf{cl}(v)$ to refer to the clock of v. By contrast, a value of type $\triangledown A$ is a delayed computation that will produce a value of type A whenever *any* clock ticks.

The universal later modality provides the interface of an applicative functor [McBride and Paterson 2008] made up of the introduction form $\mathsf{delay} : A \to \triangledown A$ and the applicative action operator $\circledast : \triangledown (A \to B) \to \triangledown A \to \triangledown B$. That is, we can delay any value into the future and apply a delayed function to a delayed argument to obtain its delayed result. In addition, $\triangledown$ also features the operator $\mathbin{\triangleright\!\!=} : \triangledown (A \to B) \to \triangleq A \to \triangleq B$, which is a variant of $\circledast$ that interacts with the existential later modality. This interaction between the two later modalities is possible since a universally delayed function $f : \triangledown (A \to B)$ is available at any time in the future. In particular, it is available when the clock $\mathsf{cl}(v)$ of an existentially delayed value $v : \triangleq A$ ticks so that v produces a value of type A. Therefore, $f \mathbin{\triangleright\!\!=} v$ produces a value whenever v does, i.e. $\mathsf{cl}(f \mathbin{\triangleright\!\!=} v) = \mathsf{cl}(v)$.

With the help of these ingredients, we can define a functorial action operator $\triangleright$, which we will use extensively in examples. It applies a function to an existentially delayed computation:

$$\frac{x : A \in \Gamma}{\Gamma \vdash_\Delta x : A} \qquad \frac{}{\Gamma \vdash_\Delta () : 1} \qquad \frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash_\Delta t : A \to B \quad \Gamma \vdash_\Delta t' : A}{\Gamma \vdash_\Delta t\, t' : B} \qquad \frac{\Gamma \vdash_\Delta t : A_i}{\Gamma \vdash_\Delta \mathsf{in}_i\, t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash_\Delta t_i : B \quad \Gamma \vdash_\Delta t : A_1 + A_2}{\Gamma \vdash_\Delta \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B} \qquad \frac{\Gamma \vdash_\Delta t : A \quad \Gamma \vdash_\Delta t' : B}{\Gamma \vdash_\Delta (t, t') : A \times B} \qquad \frac{\Gamma \vdash_\Delta t : A_1 \times A_2}{\Gamma \vdash_\Delta \pi_i\, t : A_i} \qquad \frac{\Gamma \vdash_\Delta t : F[\mu\,\alpha.F/\alpha]}{\Gamma \vdash_\Delta \mathsf{cons}_{\mu\alpha.F}\, t : \mu\,\alpha.F}$$

$$\frac{\Gamma, x : F[(\mu\alpha.F) \times A/\alpha] \vdash_\Delta s : A \quad \Gamma \vdash_\Delta t : \mu\alpha.F}{\Gamma \vdash_\Delta \mathsf{rec}(x.s, t) : A} \qquad \frac{\kappa : \mathsf{Chan}\, A \in \Delta}{\Gamma \vdash_\Delta \kappa : \mathsf{Chan}\, A} \qquad \frac{}{\Gamma \vdash_\Delta \mathsf{chan}_A : \mathsf{Chan}\, A}$$

$$\frac{\Gamma \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{delay}\, t : \circledcirc A} \qquad \frac{\Gamma \vdash_\Delta t : \circledcirc (A \to B) \quad \Gamma \vdash_\Delta t' : \circledcirc A}{\Gamma \vdash_\Delta t \circledast t' : \circledcirc B} \qquad \frac{\Gamma \vdash_\Delta t : \circledcirc (A \to B) \quad \Gamma \vdash_\Delta t' : \textcircled{\exists} A}{\Gamma \vdash_\Delta t \circledS t' : \textcircled{\exists} B}$$

$$\frac{}{\Gamma \vdash_\Delta \mathsf{never} : \textcircled{\exists} A} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Chan}\, A}{\Gamma \vdash_\Delta \mathsf{wait}\, t : \textcircled{\exists} A} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Sig}\, (A + 1)}{\Gamma \vdash_\Delta \mathsf{watch}\, t : \textcircled{\exists} A} \qquad \frac{\Gamma \vdash_\Delta t_1 : \textcircled{\exists} A_1 \quad \Gamma \vdash_\Delta t_2 : \textcircled{\exists} A_2}{\Gamma \vdash_\Delta \mathsf{select}\, t_1\, t_2 : \textcircled{\exists} ((A_1 + A_2) + (A_1 \times A_2))}$$

$$\frac{\Gamma, x : \circledcirc A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}\, x.t : A} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Sig}\, A}{\Gamma \vdash_\Delta \mathsf{head}\, t : A} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Sig}\, A}{\Gamma \vdash_\Delta \mathsf{tail}\, t : \textcircled{\exists} (\mathsf{Sig}\, A)} \qquad \frac{\Gamma \vdash_\Delta s : A \quad \Gamma \vdash_\Delta t : \textcircled{\exists} (\mathsf{Sig}\, A)}{\Gamma \vdash_\Delta s ::_A t : \mathsf{Sig}\, A}$$

Fig. 3. Typing rules. We use A, B to range over closed types as defined in Fig. 2.

$s_1 =$	1		2	3	4		⋯
$s_2 =$	6	7	8			9	⋯
$s_3 = \mathsf{tail}\ s_1 =$			2	3	4		⋯
$\mathit{map}\ (\lambda x.x + 1)\ s_1 =$	2		3	4	5		⋯
$\mathit{sample}\ s_1\ s_2 =$	(1, 6)		(2, 8)	(3, 8)	(4, 8)		⋯
$\mathit{const}\ 42 =$	42						
$\mathit{switch}\ s_2\ s_3 =$	6	7	2	3	4		⋯
$\mathit{mkSig}\ (\mathsf{wait}\ \kappa) =$			7	8			⋯
$\mathit{filter}\ \mathit{isEven}\ s_3 =$			2		4		⋯
$\mathit{scan}\ (\lambda m\ n.m + n)\ 0\ s_1 =$	1		3	6	10		⋯
$\mathit{scanAwait}\ (\lambda m\ n.m + n)\ 0\ s_3 =$	0		2	5	9		⋯
$\mathit{interleave}\ (\lambda x\ y.x)\ s_3\ (\mathsf{tail}\ s_2) =$		7	2	3	4	9	⋯
$\mathit{zip}\ s_1\ s_2 =$	(1, 6)	(1, 7)	(2, 8)	(3, 8)	(4, 8)	(4, 9)	⋯
	0	e_0	e_1	e_2	e_3	e_4	time

Fig. 4. Example signals (and delayed signals) produced by signal combinators when receiving event sequence $\tau = e_0, e_1, e_2, e_3, e_4$. For *mkSig* (wait κ), we assume that $\kappa : \mathsf{Chan}\ \mathsf{Nat} \in \Delta$, $e_1 = \kappa \mapsto 7$, and $e_2 = \kappa \mapsto 8$, whereas the other events do not concern κ.

$$\_ \triangleright \_ : (A \to B) \to \textcircled{\exists}\, A \to \textcircled{\exists}\, B$$
$$f \triangleright x = \mathsf{delay}\, f \circledS x$$

The type of signals producing values of type A is defined by guarded recursion as $\mathsf{Sig}\ A \cong A \times \textcircled{\exists}\ (\mathsf{Sig}\ A)$. Concretely, signals are constructed using the $::_A$ operator of type $A \to \textcircled{\exists}\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ A$ so that the signal $v ::_A w$ has the current value $v : A$ and will update as soon as $\mathsf{cl}(w)$ ticks. We often elide the subscript A from $::_A$ and simply write $v :: w$ when A can be inferred from the context.

Since we can think of a value w of type $\Circledcirc$ (Sig A) as a pair (θ, f) consisting of a clock θ and a delayed computation f that produces a new value of type Sig A whenever θ ticks, we can in turn think of a signal s : Sig A as a sequence $v_0 \langle\theta_0\rangle v_1 \langle\theta_1\rangle \ldots$ consisting of lazily computed values v_i of type A separated by clocks θ_i. After the clocks $\theta_0, \ldots, \theta_{i-1}$ have ticked in succession, the signal has the value v_i and waits for a tick on θ_i. The current value and the future of a signal can be accessed via head : Sig $A \to A$ and tail : Sig $A \to \Circledcirc$ (Sig A), respectively. However, we will usually use pattern matching syntax instead. For example, the *map* function can be implemented as follows:

$$map : (A \to B) \to \text{Sig } A \to \text{Sig } B$$
$$map\ f\ (x :: xs) = f\ x :: (map\ f \triangleright xs)$$

This is a *guarded recursive definition*, as the recursive call *map f* is delayed by the $\triangleright$ operator. We show how the syntactic sugar of such guarded recursive function definitions is translated into the guarded fixed point combinator fix in section 2.3.

Fig. 4 illustrates how signal combinators such as *map* work by showing the asynchronous timing of different (delayed) signals. Each row represents a signal (of type Sig A) or a delayed signal (of type $\Circledcirc$ (Sig A) and thus with no initial value at time 0). An entry v in a row indicates that the (delayed) signal updates its value to v at that time. To illustrate these (delayed) signals, we assume that we have received a sequence of events $\tau = e_0, e_1, e_2, e_3, e_4$, whose occurrences are shown at the bottom of Fig. 4. The figure illustrates the relative timing of the two signals s_1 : Sig Nat and s_2 : Sig Nat, which intuitively denote the following sequences of values and clocks:[1]

$$s_1 \approx 1 \langle\theta_0\rangle 2 \langle\theta_1\rangle 3 \langle\theta_2\rangle 4 \langle\theta_3\rangle \ldots \quad \text{and} \quad s_2 \approx 6 \langle\theta_0'\rangle 7 \langle\theta_1'\rangle 8 \langle\theta_2'\rangle 9 \langle\theta_3'\rangle \ldots$$

Fig. 4 tells us that when event e_0 occurs, s_2 produces a new value 7, whereas s_1 does not. That is, e_0 causes θ_0' to tick, but not θ_0. Similarly, e_1 causes both θ_0 and θ_1' to tick simultaneously, which in turn causes s_1 and s_2 to produce new values 2 and 8, respectively. Given a term t and a sequence of events τ, we write $t \rightsquigarrow_\tau v$ for the evaluation of t to value v in response to τ. For example, given the sequence of events e_0, e_1, e_2 from Fig. 4, we have that $s_1 \rightsquigarrow_{e_0,e_1,e_2} 3 :: d_1$ and $s_2 \rightsquigarrow_{e_0,e_1,e_2} 8 :: d_2$, for some $d_1, d_2 : \Circledcirc$ (Sig Nat) with $\mathsf{cl}(d_1) = \theta_2$ and $\mathsf{cl}(d_2) = \theta_2'$. We call $\rightsquigarrow_\tau$ the *reactive evaluation semantics* of Rizzo and define it formally in section 4.

Applying *map* $(\lambda x.x + 1)$ to s_1 produces a signal intuitively denoted as follows:

$$map\ (\lambda x.x + 1)\ s_1 \approx 2 \langle\theta_0\rangle 3 \langle\theta_1\rangle 4 \langle\theta_2\rangle 5 \langle\theta_3\rangle \ldots$$

Because $\mathsf{cl}(map\ f \triangleright xs) = \mathsf{cl}(xs)$, *map* produces a signal with the same clocks as the argument signal. For example, *map* $(\lambda x.x + 1)\ s_1 \rightsquigarrow_{e_0,e_1,e_2} 4 :: d$ for some $d : \Circledcirc$ (Sig Nat) with $\mathsf{cl}(d) = \theta_2$.

Using the *map* combinator, we can implement a sampling combinator that takes two signals and samples the value of the second signal each time the first signal updates:

$$sample : \text{Sig } A \to \text{Sig } B \to \text{Sig } (A \times B)$$
$$sample\ xs\ ys = map\ (\lambda x.(x, \text{head } ys))\ xs$$

Each time xs produces a new value v, the function $(\lambda x.(x, \text{head } ys))$ is applied to v so that the resulting signal produces a value obtained by evaluating $(v, \text{head } ys)$. Let's consider the example *sample* $s_1\ s_2$, shown in Fig. 4. Recall that e_2 causes the clock θ_1 of s_1 to tick, and that $s_1 \rightsquigarrow_{e_0,e_1,e_2} 3 :: d_1$ and $s_2 \rightsquigarrow_{e_0,e_1,e_2} 8 :: d_2$. Hence, when e_2 occurs, *sample* $s_1\ s_2$ will sample s_2, which has been evaluated to $8 :: d_2$. Therefore, *sample* $s_1\ s_2 \rightsquigarrow_{e_0,e_1,e_2} (3, 8) :: d'$ for some d' with $\mathsf{cl}(d') = \theta_2$.

[1] More accurately, the sequence $v_0 \langle\theta_0\rangle v_1 \langle\theta_1\rangle \ldots$ denoted by a signal s is w.r.t. some given sequence of events $(e_i)_{i\in\mathbb{N}}$. Specifically, each v_i and θ_i are obtained from $s \rightsquigarrow_{e_0,\ldots,e_n} v_i :: w_i$ with $\theta_i = \mathsf{cl}(w_i)$ for some prefix $e_0, \ldots, e_n$ of $(e_i)_{i\in\mathbb{N}}$.

2.2 Constructing Delayed Computations

In addition to delay, $\circledast$, and $\circledcirc$, Rizzo features four more primitives to construct delayed computations, namely never, select, wait, and watch. We discuss each of them in turn below and illustrate them with signal combinators and accompanying examples in Fig. 4.

The primitive never : $\Diamond\!\!\!\!-\, A$ allows us to construct delayed computations with a clock that will never tick. This is useful for constructing constant signals:

$$\begin{aligned}&\mathit{const} : A \to \mathsf{Sig}\ A\\ &\mathit{const}\ x = x :: \mathsf{never}\end{aligned}$$

That is, intuitively speaking, *const* v denotes a finite sequence $v\ \langle\theta\rangle$ with a clock θ that never ticks.

The primitive select takes two delayed computations $v : \Diamond\!\!\!\!-\, A$, $w : \Diamond\!\!\!\!-\, B$ and produces a delayed computation of type $\Diamond\!\!\!\!-\, ((A + B) + (A \times B))$ with a clock that ticks whenever $\mathsf{cl}(v)$ or $\mathsf{cl}(w)$ ticks. If $\mathsf{cl}(v)$ ticks first, then select $v\ w$ produces the value produced by v. Likewise, if $\mathsf{cl}(w)$ ticks first, then select $v\ w$ produces the value produced by w. If both $\mathsf{cl}(v)$ and $\mathsf{cl}(w)$ tick simultaneously, then select $v\ w$ produces a pair of values with each component drawn from the corresponding result of v and w. The sum type $(A + B) + (A \times B)$ witnesses these three contingencies.

To make the type of select more readable, we use the shorthand *Select* $A\ B$ for the type $(A + B) + (A \times B)$ as well as the shorthands `left` s, `right` t, and `both` $s\ t$ for $\mathsf{in}_1\ (\mathsf{in}_1\ s)$, $\mathsf{in}_1\ (\mathsf{in}_2\ t)$, and $\mathsf{in}_2\ (s, t)$, respectively. We can use select to construct dynamic dataflows. The prototypical signal combinator with dynamic dataflow is *switch*, which constructs a signal that initially behaves like the first signal it is given but then switches its behaviour to the second – delayed – signal as soon as that second signal arrives:

$$\begin{aligned}&\mathit{switch} : \mathsf{Sig}\ A \to \Diamond\!\!\!\!-\, (\mathsf{Sig}\ A) \to \mathsf{Sig}\ A\\ &\mathit{switch}\ (x :: xs)\ d = x :: (\mathit{cont} \triangleright \mathsf{select}\ xs\ d)\\ &\quad \textbf{where}\ \mathit{cont} : \mathit{Select}\ (\mathsf{Sig}\ A)\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ A\\ &\qquad\quad \mathit{cont}\ (\mathtt{left}\ xs'\ \ ) = \mathit{switch}\ xs'\ d\\ &\qquad\quad \mathit{cont}\ (\mathtt{right}\ \ d') = d'\\ &\qquad\quad \mathit{cont}\ (\mathtt{both}\ \_\ d') = d'\end{aligned}$$

As we can see, *switch* uses select to check which of xs and d arrives first. If xs arrives first and produces a new signal $xs' : \mathsf{Sig}\ A$, then *switch* recursively continues. Otherwise, if d arrives either before or simultaneously with xs, then *switch* simply returns the resulting signal $d' : \mathsf{Sig}\ A$.

To understand the wait primitive, we must first consider what kind of events a Rizzo program can receive and how these events cause a clock to tick. A Rizzo program receives inputs from its environment via *channels*. A channel of type $\mathsf{Chan}\ A$ can receive data of type A. We write $\kappa \mapsto t$ for the *event* that the channel κ has received input t, which is typically a value but does not need to be one. For example, if the channel context Δ contains a channel $\kappa_{\mathsf{keyboard}} : \mathsf{Chan}\ \mathit{Char}$ that receives the characters typed on the keyboard, then the event $\kappa_{\mathsf{keyboard}} \mapsto \mathtt{'a'}$ indicates that the user has pressed the 'a' key. We can wait to receive a value from a channel using $\mathit{wait} : \mathsf{Chan}\ A \to \Diamond\!\!\!\!-\, A$, which takes a channel $\kappa : \mathsf{Chan}\ A$ and constructs a delayed computation with a clock that ticks whenever an event $\kappa \mapsto t$ occurs, which then causes the delayed computation to produce the value v that t evaluates to. For example, $\mathit{wait}\ \kappa_{\mathsf{keyboard}} : \Diamond\!\!\!\!-\, \mathit{Char}$ is a delayed computation that produces a character value as soon as the user presses a key. In turn, we can use delayed computations to construct signals using the following combinator:

$$\begin{aligned}&\mathit{mkSig} : \Diamond\!\!\!\!-\, A \to \Diamond\!\!\!\!-\, (\mathsf{Sig}\ A)\\ &\mathit{mkSig}\ da = (\lambda a.a :: \mathit{mkSig}\ da) \triangleright da\end{aligned}$$

Fig. 4 illustrates the delayed signal *mkSig* (wait κ) under the assumption that the events in τ are more specifically $e_1 = \kappa \mapsto 7$ and $e_2 = \kappa \mapsto 8$, while the events e_0, e_3, e_4 do not concern κ, i.e. they are of the form $\kappa' \mapsto t$ for some $\kappa' \neq \kappa$.

Channels are either drawn from the channel context Δ or can be constructed by the program using $\mathsf{chan}_A : \mathsf{Chan}\ A$. A channel $\kappa : \mathsf{Chan}\ A \in \Delta$ is a built-in channel like $\kappa_{\mathsf{keyboard}} : \mathsf{Chan}\ \mathit{Char}$ or $\kappa_{\mathsf{mouse}} : \mathsf{Chan}\ (\mathit{Int} \times \mathit{Int})$, whereas dynamically created channels can be used to construct objects that may receive input. For example, in a GUI, we may create a channel $\kappa : \mathsf{Chan}\ 1$ whenever we construct a button, so that we may receive an event $\kappa \mapsto ()$ every time the button is pressed.

Finally, the primitive $\mathsf{watch} : \mathsf{Sig}\ (A + 1) \to ⦵\ A$ allows us to treat a signal of type $\mathsf{Sig}\ (A + 1)$ as an *internal* source of events. We typically call a signal of this type a *partial signal*. For readability, we use the shorthand *Maybe* A for $A + 1$, and we write $\mathtt{just}\ t$ and $\mathtt{nothing}$ for $\mathsf{in}_1\ t$ and $\mathsf{in}_2\ ()$, respectively. Given a partial signal $s : \mathsf{Sig}\ (\mathit{Maybe}\ A)$, $\mathsf{watch}\ s$ constructs a delayed computation that produces a value of type A whenever s is updated to a value of the form $\mathtt{just}\ v$ for some $v : A$. With the help of watch, we can implement *filter*, which takes a predicate p and a delayed signal s to construct a new delayed signal that contains only those elements of s that satisfy p:

$$\mathit{filter} : (A \to \mathit{Bool}) \to ⦵\ (\mathsf{Sig}\ A) \to ⦵\ (\mathsf{Sig}\ A)$$
$$\mathit{filter}\ p\ s = \mathit{mkSig}\ (\mathsf{watch}\ (\mathtt{nothing} :: (\mathit{mapMaybe}\ p\ s)))$$

$$\mathit{mapMaybe} : (A \to \mathit{Bool}) \to ⦵\ (\mathsf{Sig}\ A) \to ⦵\ (\mathsf{Sig}\ (\mathit{Maybe}\ A))$$
$$\mathit{mapMaybe}\ p\ d = \mathit{map}\ (\lambda x.\mathbf{if}\ p\ x\ \mathbf{then}\ \mathtt{just}\ x\ \mathbf{else}\ \mathtt{nothing}) \triangleright d$$

We use *mapMaybe* to turn a delayed signal d into a delayed *partial* signal that contains only the elements of d that satisfy the predicate p. Then *filter* uses watch on the partial signal to construct the desired (non-partial) signal.

Now that we have seen all primitive operations to construct delayed computations, we can give a precise account of what clocks are: A clock θ is a finite set containing channels $\kappa : \mathsf{Chan}\ A$ and partial signals $s : \mathsf{Sig}\ (\mathit{Maybe}\ A)$. Given an event $\kappa \mapsto t$, the clock θ ticks iff $\kappa \in \theta$ or there is a partial signal $s \in \theta$ such that $\kappa \mapsto t$ causes s to produce a new value of the form $\mathtt{just}\ w$. In particular, we define $\mathsf{cl}(\mathsf{never}) = \emptyset$, i.e. $\mathsf{cl}(\mathsf{never})$ never ticks; $\mathsf{cl}(\mathsf{wait}\ \kappa) = \{\kappa\}$; $\mathsf{cl}(\mathsf{watch}\ v) = \{v\}$; $\mathsf{cl}(v \circledcirc w) = \mathsf{cl}(w)$; and $\mathsf{cl}(\mathsf{select}\ v\ w) = \mathsf{cl}(v) \cup \mathsf{cl}(w)$, i.e. $\mathsf{cl}(\mathsf{select}\ v\ w)$ ticks whenever $\mathsf{cl}(v)$ or $\mathsf{cl}(w)$ ticks. In addition, we also extend the notion of clocks to non-value terms t by defining $\mathsf{cl}(t) = \mathsf{cl}(v)$, where v is the result of evaluating t with the empty sequence of events, i.e. $t \rightsquigarrow . \ v$.

2.3 Recursive Types and Guarded Recursion

Rizzo features recursive types of the form $\mu\alpha.F$, where α may not appear in the domain of a function type nor in the scope of Chan, ⦵, or ⓥ. This restriction can be seen in the type formation rules in Fig. 2, which permit the types $F \to G$, $\mathsf{Chan}\ F$, $⦵F$, and $ⓥF$ only if there are no free type variable occurrences in F. Supported recursive types include standard inductive types such as the type of natural numbers $\mathsf{Nat} = \mu\alpha.1 + \alpha$, lists $\mathsf{List}\ A = \mu\alpha.1 + (A \times \alpha)$, binary trees $\mu\alpha.A + (\alpha \times A \times \alpha)$, and infinitely branching trees $\mu\alpha.A + (A \times (\mathsf{Nat} \to \alpha))$, but also nested inductive types such as the type of rose trees $\mu\alpha.A \times \mathsf{List}\ \alpha$. Importantly, we also support recursive types with signals, such as a type of binary trees $\mu\alpha.A + \mathsf{Sig}\ (\alpha \times A \times \alpha)$, where each inner node is a signal and can thus change over time. We will see an example of this in section 3, where we use such trees to represent graphical user interfaces that can change dynamically in response to user input.

We can traverse such recursive data structures using the primitive recursion combinator rec, but in examples we use standard pattern matching and recursion syntax, which can be translated into uses of rec. For instance, consider the following recursive definitions:

$$\begin{array}{ll}\textbf{data}\ \mathsf{List}\ A = \mathtt{nil} & \mathit{length} : \mathsf{List}\ A \to \mathsf{Nat} \\ \quad \mid \mathtt{cons}\ A\ (\mathsf{List}\ A) & \mathit{length}\ \mathtt{nil} \quad\quad\ = 0 \\ & \mathit{length}\ (\mathtt{cons}\ x\ xs) = 1 + \mathit{length}\ xs\end{array}$$

This recursion and pattern matching syntax desugars into the following Rizzo type and term:

$$\mathsf{List}\ A = \mu\alpha.1 + (A \times \alpha) \qquad \mathit{length} = \lambda l.\mathsf{rec}(r.\mathsf{case}\ r\ \mathsf{of}\ \mathsf{in}_1\ x.0; \mathsf{in}_2\ x.1 + \pi_2\ (\pi_2\ x), l)$$

In addition, Rizzo features guarded recursion [Nakano 2000] via the guarded fixed point combinator fix. Instead of a general fixed point combinator of type $(A \to A) \to A$, the guarded fixed point combinator has type $(\textcircled{\scriptsize▽}\ A \to A) \to A$. Following Bahr and Møgelberg [2023], we make the argument have type $\textcircled{\scriptsize▽}\ A$, so that the guarded fixed point can only be unfolded in response to a future event, which in turn allows us to prove productivity of the language.

We have already seen several examples of functions defined using guarded recursion, including *map*, *mkSig*, and *switch*. If all recursive calls are guarded by an application of delay, we can translate a recursive definition from recursion syntax into a guarded fixed point. For example, the recursive call in the definition of *map* appears as the first argument to $\triangleright$ and is thus guarded by a delay. We can translate this definition into an explicit use of fix and also translate the pattern matching syntax into explicit projections with head and tail so that we obtain a term in the core calculus:

$$\begin{array}{l}\mathit{map} : (A \to B) \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\ \mathit{map} = \mathsf{fix}\ r.\lambda f.\lambda s.\textbf{let}\ x = \mathsf{head}\ s\ \textbf{in let}\ xs = \mathsf{tail}\ s\ \textbf{in}\ f\ x :: (\mathsf{delay}\ (\lambda r'.r'\ f) \circledast r \circledcirc xs)\end{array}$$

For readability, we still keep the standard syntactic sugar $\textbf{let}\ x = s\ \textbf{in}\ t$ as notation for $(\lambda x.t)\ s$.

This translation from recursive surface syntax to explicit guarded fixed points follows a general scheme. Given a recursively defined function f where all recursive calls occur under a delay, we can desugar a recursive definition of the form $f x_1 \ldots x_n = K[\mathsf{delay}\ t_1, \ldots, \mathsf{delay}\ t_m]$, where f does not occur in K, into the following form that uses fix:

$$f = \mathsf{fix}\, r.\lambda x_1 \ldots \lambda x_n.K[\mathsf{delay}(\lambda r'.t_1[r'/f]) \circledast r, \ldots, \mathsf{delay}(\lambda r'.t_m[r'/f]) \circledast r]$$

Moreover, pattern matching is desugared into corresponding eliminators of the calculus in the standard way as we have seen in the example for *map* above.

2.4 Equational Reasoning Limitations

The forgetful semantics of signals means that β and η conversions are not equivalence preserving in general. For example, $(\lambda x.\mathsf{delay}\ x)\ (\mathsf{head}\ xs)$ is semantically different from its β-contractum $\mathsf{delay}\ (\mathsf{head}\ xs)$, and $f\ (\mathsf{head}\ xs)$ is semantically different from its η-expansion $\lambda x.f\ (\mathsf{head}\ xs)\ x$. The head of a signal may be different depending on whether it is evaluated now or in the future, because evaluating a signal erases its past to avoid space leaks by construction. As an example, consider a β-expanded version of *sample*, defined by $\mathit{sample}_\beta\ xs\ ys = \mathit{map}\ ((\lambda y\ x.(x, y))\ (\mathsf{head}\ ys))\ xs$. Given s_1, s_2 from Fig. 4, $\mathit{sample}\ s_1\ s_2$ produces $(1, 6)\ \langle\theta_0\rangle\ (2, 8)\ \langle\theta_1\rangle\ (3, 8)\ \langle\theta_2\rangle\ (4, 8)\ \langle\theta_3\rangle\ \ldots$, whereas $\mathit{sample}_\beta\ s_1\ s_2$ produces $(1, 6)\ \langle\theta_0\rangle\ (2, 6)\ \langle\theta_1\rangle\ (3, 6)\ \langle\theta_2\rangle\ (4, 6)\ \langle\theta_3\rangle\ \ldots$, i.e. the latter does not sample s_2 but simply returns the initial value of s_2, namely 6.

A similar situation also appears in Async RaTT [Bahr and Møgelberg 2023] (the only other asynchronous modal FRP calculus with space leaks guarantees), where the term $\mathsf{read}_\kappa : A$ reads the current value of a channel κ and thus evaluates to different values depending on *when* it is evaluated. Therefore, similarly to Async RaTT, equational reasoning in Rizzo has to restrict β- and η-conversion to values, i.e. $(\lambda x.t)\ v = t[v/x]$ and $v = \lambda x.v\ x$.

$$
\begin{array}{ll}
\mathit{map} & : (A \to B) \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\
\mathit{mkSig} & : \Diamond\ A \to \Diamond\ (\mathsf{Sig}\ A) \\
\mathit{const} & : A \to \mathsf{Sig}\ A \\
\mathit{sample} & : \mathsf{Sig}\ A \to \mathsf{Sig}\ B \to \mathsf{Sig}\ (A \times B) \\
\mathit{zip} & : \mathsf{Sig}\ A \to \mathsf{Sig}\ B \to \mathsf{Sig}\ (A \times B)
\end{array}
\qquad
\begin{array}{ll}
\mathit{switch} & : \mathsf{Sig}\ A \to \Diamond\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ A \\
\mathit{interleave} & : (A \to A \to A) \to \Diamond\ (\mathsf{Sig}\ A) \to \Diamond\ (\mathsf{Sig}\ A) \to \Diamond\ (\mathsf{Sig}\ A) \\
\mathit{scan} & : (B \to A \to B) \to B \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\
\mathit{scanAwait} & : (B \to A \to B) \to B \to \Diamond\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ B \\
\mathit{filter} & : (A \to \mathit{Bool}) \to \Diamond\ (\mathsf{Sig}\ A) \to \Diamond\ (\mathsf{Sig}\ A)
\end{array}
$$

Fig. 5. Small FRP library for signal processing.

3 Programming in Rizzo

In this section, we present a collection of example Rizzo programs to illustrate the expressiveness of the language and to compare it with previous work. We start by implementing a small library of signal combinators and then use some of these combinators to implement a simple GUI library along with a small GUI application.

3.1 FRP Library

Fig. 5 lists the type signatures of a small signal combinator library, which we have already partly implemented in section 2. We discuss the implementation of the remaining combinators below.

First, consider the *scan* combinator, which is a variant of the *map* combinator where the produced output may depend on the previous value of the output signal:

$$
\begin{array}{l}
\mathit{scan} : (B \to A \to B) \to B \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\
\mathit{scan}\ f\ b\ (a :: as) = \mathbf{let}\ b' = f\ b\ a\ \mathbf{in}\ b' :: (\mathit{scan}\ f\ b' \rhd as)
\end{array}
$$

The example in Fig. 4 uses *scan* to sum up the numbers produced by signal s_1.

We can derive a variant of *scan* that takes a delayed signal as argument:

$$
\begin{array}{l}
\mathit{scanAwait} : (B \to A \to B) \to B \to \Diamond\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ B \\
\mathit{scanAwait}\ f\ b\ s = b :: (\mathit{scan}\ f\ b \rhd s)
\end{array}
$$

The *zip* combinator has the same type as *sample*, but unlike the latter, the signal produced by $\mathit{zip}\ as\ bs$ updates whenever as or bs updates, i.e. $\mathsf{cl}(\mathsf{tail}\ (\mathit{zip}\ as\ bs)) = \mathsf{cl}(\mathsf{tail}\ as) \cup \mathsf{cl}(\mathsf{tail}\ bs)$:

$$
\begin{array}{l}
\mathit{zip} : \mathsf{Sig}\ A \to \mathsf{Sig}\ B \to \mathsf{Sig}\ (A \times B) \\
\mathit{zip}\ as\ bs = (\mathsf{head}\ as, \mathsf{head}\ bs) :: (\mathit{cont} \rhd \mathsf{select}\ (\mathsf{tail}\ as)\ (\mathsf{tail}\ bs)) \\
\quad \mathbf{where}\ \mathit{cont}\ (\mathtt{left}\ as') \quad\ = \mathit{zip}\ as'\ bs \\
\qquad\quad \mathit{cont}\ (\mathtt{right}\ bs') \quad = \mathit{zip}\ as\ bs' \\
\qquad\quad \mathit{cont}\ (\mathtt{both}\ as'\ bs') = \mathit{zip}\ as'\ bs'
\end{array}
$$

That is, at any given time, the signal $\mathit{zip}\ as\ bs$ has the current value (a, b), where a is the current value of as and b is the current value of bs.

If we have two *delayed* signals of the same type, we can also combine them by interleaving, so that each update on the resulting delayed signal is an update from one of the two original delayed signals. This idea is implemented by the *interleave* combinator, which also takes a function that serves as a tiebreaker for when both delayed signals update at the same time:

$$
\begin{array}{l}
\mathit{interleave} : (A \to A \to A) \to \Diamond\ (\mathsf{Sig}\ A) \to \Diamond\ (\mathsf{Sig}\ A) \to \Diamond\ (\mathsf{Sig}\ A) \\
\mathit{interleave}\ f\ xs\ ys = \mathit{cont} \rhd \mathsf{select}\ xs\ ys \\
\quad \mathbf{where}\ \mathit{cont}\ (\mathtt{left}\ (x :: xs')) \qquad\quad = x \quad\ :: \mathit{interleave}\ f\ xs'\ ys \\
\qquad\quad \mathit{cont}\ (\mathtt{right}\ \ (y :: ys')) = y \quad\ :: \mathit{interleave}\ f\ xs\ \ ys' \\
\qquad\quad \mathit{cont}\ (\mathtt{both}\ (x :: xs')\ (y :: ys')) = f\ x\ y :: \mathit{interleave}\ f\ xs'\ ys'
\end{array}
$$

```
data Button    = mkButton (Sig String) (Sig Colour) (Chan 1)
data TextField = mkTextField (Sig String) (Sig Colour) (Chan String)
data Layout    = horizontal | vertical
data Widget    = button Button | textfield TextField | stack Layout (Sig (List Widget))

simpleButton : String → Button
simpleButton txt = mkButton (const txt) (const Black) chan

onClick : Button → ∃ (Sig 1)
onClick (mkButton _ _ k) = mkSig (wait k)

simpleTF : String → TextField
simpleTF txt = let k = chan in mkTextField (txt :: mkSig (wait k)) (const Black) k
```

Fig. 6. Simple GUI library.

3.2 GUI Application

Fig. 6 implements the interface for a very simple GUI library. It defines a *Widget* type to represent the hierarchical structure of a GUI consisting of two basic widget types – buttons and text fields – and stack widgets that can place several widgets next to each other (horizontally or vertically).

The attributes of widgets, such as their colour, are described using signals, so that these may change over time. In addition, widgets may produce events, e.g. when a button is pressed or the user types text into a text field. Such events and their accompanying data are sent on channels: A button has a channel of type Chan 1, so that when the button is pressed, the button's channel receives a unit value; and the text field has a channel of type Chan *String*, so that when the text field's contents change, we receive the new contents on that channel.

For example, we may construct a button that changes its colour from black to red when pressed:

```
colourButton : Button
colourButton = let c = chan in mkButton (const "click me") (Black :: (const Red @ wait c)) c

_ @ _ : B → ∃ A → ∃ B
v @ d = (λ_.v) ▷ d
```

We define the @ operator so that we can delay the constant signal *const Red* until wait *c* arrives.

Fig. 6 also defines some helper functions to construct simple widgets: a button with a constant text and a fixed colour; and a text field that displays exactly the text that the user has typed. Let's use these simple building blocks to implement a small GUI with an 'Add' button, which allows the user to add an extra text field to the GUI with each press of the button:

```
btn : Button
btn = simpleButton "Add"

newField : 1 → Widget
newField _ = textfield (simpleTF "")

newWidgets : ∃ (Sig Widget)
newWidgets = map newField ▷ onClick btn

allWidgets : Sig (List Widget)
allWidgets = scanAwait snoc [button btn] newWidgets

gui : Widget
gui = stack vertical allWidgets
```

This example illustrates the highly dynamic structure that is enabled by the nesting of signals in the *Widget* type. The GUI's behaviour is completely described by *gui* of type *Widget*: It describes the tree-structured hierarchy of widgets that are supposed to be presented to the user, and how this hierarchy of widgets is supposed to change in response to input from the user. The user's interaction with the GUI produces values that are sent on the channels associated with individual

widgets. In particular, each time the button *btn* is pressed, an event occurs on *btn*'s channel, which causes the delayed signal $\mathit{newWidgets} : ⧆ (\mathsf{Sig}\ \mathit{Widget})$ to produce a new text field widget. In turn, this delayed signal of new widgets is collected by *allWidgets* into a signal of all widgets that have been created starting with the button *btn*. To this end, we use *scanAwait* to add any new widget from *newWidgets* to the initial list of widgets [`button` *btn*] via $\mathit{snoc} : \mathsf{List}\ A \to A \to \mathsf{List}\ A$ that appends an element to the end of a list.

To demonstrate the highly dynamic nature of widgets enabled by their nested-signal structure, we revise the above example GUI to also allow the removal of widgets. To this end, we change the definition of the *newField* function so that in addition to a text field, it also adds a 'Remove' button. In turn, the 'Remove' button allows the user to remove the text field and the 'Remove' button itself.

```
newField : 1 → Widget
newField _ = let remove   = simpleButton "Remove" in
             let tfAndBtn = [textfield (simpleTF ""), button remove]
             in stack horizontal (tfAndBtn :: (const nil @ onClick remove))
```

The function adds a stack containing a text field and a 'Remove' button, but as soon as the 'Remove' button is clicked, the stack contents are changed to the empty list of widgets `nil`.

3.3 Comparison to Async RaTT

As mentioned in the introduction, Rizzo takes inspiration from the Async RaTT calculus of Bahr and Møgelberg [2023] but is simpler and more expressive. With the help of the examples in the previous sections, we can explore the differences in more detail.

3.3.1 Expressiveness. Like Rizzo, Async RaTT also features two later modalities (⧆ and ⧊) and a notion of clocks. However, in order to obtain guarantees about space leaks, Async RaTT also features the stable modality □ and a Fitch-style type system that restricts the scope of variables unless they are of a stable type. A type A is stable if all occurrences of $\to$, Sig, and ⧆ in A are guarded by □. As a result, functions, signals, and delayed computations cannot be moved across time. For example, *map* is of type $\Box(A \to B) \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B$ in Async RaTT, i.e. the function must be boxed with the □ modality so that *map* can use it at any time in the future. This limits what kind of functions we may give *map*. For example, the function we pass to *map* may not contain a signal in its closure, as it does in our definition of *sample* in Rizzo.

All combinators in Fig. 5 apart from *filter* and *sample* are expressible in Async RaTT as well, but most of them with additional restrictions: All function arguments have to be boxed with □ similarly to the function argument to *map*. Additionally, *zip* requires A and B to be stable, and *scan* and *scanAwait* require B to be stable. This means that these combinators cannot be used when dealing with nested signals, which are ubiquitous when implementing GUIs as we have seen with the *Widget* type in section 3.2. Indeed, our use of *scanAwait* to implement *allWidgets* would not be allowed in Async RaTT as $\mathsf{List}\ \mathit{Widget}$ is not a stable type.

The combinators *sample* and *filter* are not expressible in Async RaTT at all due to two fundamental differences in the notion of clocks: First, Rizzo clocks only indicate timing dependencies but not data dependencies, while Async RaTT clocks appear to indicate both timing and data dependencies. For example, *sample xs ys* has a data dependency on both *xs* and *ys* as the values produced by *sample xs ys* depend on the values produced by *xs* and *ys*. However, *sample xs ys* has a timing dependency on only *xs* as $\mathsf{cl}(\mathsf{tail}\ (\mathit{sample}\ \mathit{xs}\ \mathit{ys})) = \mathsf{cl}(\mathsf{tail}\ \mathit{xs})$. Async RaTT's apparent conflation of timing and data dependencies means that Async RaTT cannot express dataflow graphs where data and timing dependencies do not coincide. Second, Rizzo has a more expressive notion of clocks,

which consist of channels (just like Async RaTT) and partial signals (not available in Async RaTT clocks). The latter enables the implementation of *filter*.

The less restrictive type system of Rizzo also makes some programs more modular and easier to express than in Async RaTT. For example, consider the following combinator in Rizzo:

$$\mathit{switchAt} : \mathsf{Sig}\ C \to \mathsf{Sig}\ C \to \Diamond\!\!\!-\ 1 \to \mathsf{Sig}\ C$$
$$\mathit{switchAt}\ \mathit{xs}\ \mathit{ys}\ d = \mathit{switch}\ \mathit{xs}\ (\mathit{ys}\ @\ d)$$

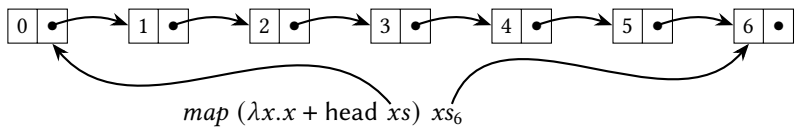
It produces a signal that first behaves like *xs* but switches to *ys* as soon as $\mathsf{cl}(d)$ ticks. While *switchAt* can be implemented in Async RaTT, it is restricted to stable types C. Moreover, the Async RaTT implementation is much more complicated: Async RaTT can express $@ : A \to \Diamond\!\!\!-\ B \to \Diamond\!\!\!-\ A$, but only if A is stable, and thus $@$ cannot be used for $\mathit{ys} : \mathsf{Sig}\ C$, even if C is stable. Therefore, *switchAt* has to be implemented as a guarded recursive function that performs a three-way synchronisation of tail *xs*, tail *ys*, and d that requires complicated nested uses of select (see Appendix B in the supplementary material). This is an example of a more general pattern often encountered in Async RaTT: By disallowing direct access to signals in any future computational context (e.g. access to $\mathit{ys} : \mathsf{Sig}\ A$ when $d : \Diamond\!\!\!-\ 1$ has ticked), Async RaTT forces us to manually keep track of the updates of each signal we want to access in a future context. In general, keeping track of n such signals can lead to $O(2^n)$ case distinctions due to nested uses of select and also restricts signals to stable payload types, which in turn precludes some use cases like GUIs as mentioned above.

Finally, while Async RaTT does not feature Rizzo's first-class channels, it would not be difficult to add them. Likewise, Async RaTT distinguishes between push and pull channels, which would not be difficult to add to Rizzo. Async RaTT features general guarded recursive types, while Rizzo only has Sig. However, Async RaTT's operational semantics only produces outputs for signals (whereas Rizzo produces outputs for any type), and in fact none of the examples presented by Bahr and Møgelberg [2023] use a guarded recursive type other than Sig.

3.3.2 Preventing Space Leaks. The difference in expressiveness between Rizzo and Async RaTT is due to their differing strategies to avoid space leaks. As discussed above, Async RaTT requires the *map* function to take a boxed function. Otherwise, Async RaTT could express the following:

$$\mathit{addHead} : \mathsf{Sig\ Nat} \to \mathsf{Sig\ Nat}$$
$$\mathit{addHead}\ \mathit{xs} = \mathit{map}\ (\lambda x.x + \mathsf{head}\ \mathit{xs})\ \mathit{xs}$$

If allowed in Async RaTT, *addHead* would have a space leak, because it has to keep the entire history of the input signal *xs* in memory. Each time a new number is received on the input signal *xs*, that number has to be stored indefinitely. The problem is that *map* recursively traverses the signal *xs*, i.e. it traverses forward in the signal as new inputs arrive, while the function closure for $\lambda x.x + \mathsf{head}\ \mathit{xs}$ contains *xs*, which points to the very beginning of the entire history of the input signal. To illustrate this, let's assume an input signal *xs* of consecutive numbers starting from 0. After receiving 7 values on *xs*, the memory representation of the input signal would look like this:



The recursively defined *map* function traverses the signal *xs* by one step each time *xs* receives a new value. After *xs* has received the number 6, we are at the 6th recursive call of *map* (illustrated above) on the signal xs_6 that starts with the number 6. However, the variable *xs* in the function closure still points to the initial signal starting at 0. Hence, we have to keep the entire prefix of the signal from 0 to 6 in memory. The longer the program runs, the more memory it will use.

Async RaTT avoids the above situation by rejecting *addHead* as ill-typed. By contrast, Rizzo avoids this situation by choosing a different semantics that does not allow signals to store their past. If *addHead* were implemented in Rizzo, *xs* would instead point at the same signal as xs_6.

4 Operational Semantics and Operational Properties

The purpose of this section is to give a precise account of the operational guarantees provided by Rizzo. To this end, we first give a formal definition of the operational semantics of Rizzo and then state the operational guarantees in terms of this formal operational semantics. Specifically, we give a formal account of the *reactive evaluation semantics* $\rightsquigarrow_\tau$ that we used informally in section 2. Given a closed Rizzo term $\vdash_\Delta t : A$, a sequence of events τ causes the pair $\langle t; \Delta\rangle$ to evaluate to the pair $\langle v; \Delta'\rangle$, denoted $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$. Since t may allocate channels via chan_A, we may obtain a new channel context Δ' that extends the original Δ. When the channel context is not relevant (e.g. when t does not use chan_A), we write $t \rightsquigarrow_\tau v$ to mean that $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$ for some Δ'.

We give an operational semantics with the following three characteristics: First, it has *by construction* no space leaks, i.e. old inputs are not kept in memory implicitly. Second, it computes its result incrementally, i.e. given $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$ and an additional event e, the operational semantics effectively computes $\langle t; \Delta\rangle \rightsquigarrow_{\tau,e} \langle w; \Delta''\rangle$ from $\langle v; \Delta'\rangle$ and e alone. Third, the operational semantics computes w by performing in-place updates of the values that changed within v.

To obtain such an operational semantics, we define a machine that represents the value v produced by $\rightsquigarrow_\tau$ as a pair $\langle p; \eta\rangle$, where p is a *machine value* that represents each signal in v as a pointer to a location l in the *heap* η that stores the signal. These machine values are defined as follows:

$$\begin{aligned} p, q \ ::= \ & () \mid \lambda x.t \mid (p, q) \mid \mathsf{in}_i\, p \mid l \mid \mathsf{tail}\, l \mid \kappa \mid \mathsf{wait}\, \kappa \mid \mathsf{watch}\, l \mid p \circledcirc q \mid \mathsf{delay}\, t \\ & \mid \ \mathsf{never} \mid \mathsf{select}\, p\, q \mid \mathsf{cons}_{\mu\alpha.F}\, q \end{aligned}$$

That is, machine values p, q only differ from values v, w (as defined in Fig. 1) in the inclusion of l and $\mathsf{tail}\, l$ and the exclusion of $v ::_A w$. The intuition is that if the heap η stores the representation of a signal $v ::_A w$ at l, then the machine value l represents the signal $v ::_A w$ and the machine value $\mathsf{tail}\, l$ represents the delayed signal w.

Using this representation of values, we define a machine that performs the reactive evaluation $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$ in incremental steps, with each such step performing an update in response to a single event in τ. Given $\tau = e_0, e_1, \ldots, e_{n-1}$, the machine first performs an initialisation step $\overset{\mathsf{init}}{\Longrightarrow}$ and then successively performs a reactive step $\overset{e_i}{\Longrightarrow}$ for each event e_i:

$$\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta_0 / \Delta_0\rangle \overset{e_0}{\Longrightarrow} \langle p; \eta_1 / \Delta_1\rangle \overset{e_1}{\Longrightarrow} \quad \ldots \quad \overset{e_{n-1}}{\Longrightarrow} \langle p; \eta_n / \Delta_n\rangle$$

The machine's state $\langle p; \eta_i / \Delta_i\rangle$ consists of the value p that t has been evaluated to, a heap η_i that stores signals, and a channel context Δ_i. The value p contains references into the heap η_i and does not change after the initialisation step. In particular, the machine evaluates terms of the form $s ::_A s'$ by evaluating the terms s and s' to values q and q', respectively, and then returning a reference l that points to a location in η_i where the signal $q ::_A q'$ is stored. Signals are updated in place since the machine only changes the heap η_i. Each pair $\langle p; \eta_i\rangle$ represents the value v_i defined by $v_i = p[\eta_i]$, which replaces each reference l in p with the signal stored at l in η_i. In particular, $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle p[\eta_n]; \Delta_n\rangle$. In the sections that follow, we give a more precise definition of the reactive evaluation semantics $\rightsquigarrow_\tau$ and the underlying machine.

4.1 Evaluation Semantics

The machine performs a step $\langle p; \eta / \Delta\rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta'\rangle$ in reaction to an event e and updates the signals stored in the heap η resulting in a new heap η'. Along the way, the machine needs to be able

Evaluation semantics $\langle t;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle$

$$\frac{}{\langle p;\varepsilon\rangle \Downarrow \langle p;\varepsilon\rangle} \qquad \frac{\langle s;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle \quad \langle t;\varepsilon'\rangle \Downarrow \langle q;\varepsilon''\rangle}{\langle (s,t);\varepsilon\rangle \Downarrow \langle (p,q);\varepsilon''\rangle} \qquad \frac{\langle t;\varepsilon\rangle \Downarrow \langle (p_1,p_2);\varepsilon'\rangle \quad i\in\{1,2\}}{\langle \pi_i\, t;\varepsilon\rangle \Downarrow \langle p_i;\varepsilon'\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle \quad i\in\{1,2\}}{\langle \mathsf{in}_i\, t;\varepsilon\rangle \Downarrow \langle \mathsf{in}_i\, p;\varepsilon'\rangle} \qquad \frac{\langle t;\varepsilon\rangle \Downarrow \langle \mathsf{in}_i\, p;\varepsilon'\rangle \quad \langle t_i[p/x];\varepsilon'\rangle \Downarrow \langle p_i;\varepsilon''\rangle \quad i\in\{1,2\}}{\langle \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1;\mathsf{in}_2\, x.t_2;\varepsilon\rangle \Downarrow \langle p_i;\varepsilon''\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \langle \lambda x.s;\varepsilon'\rangle \quad \langle t';\varepsilon'\rangle \Downarrow \langle p;\varepsilon''\rangle \quad \langle s[p/x];\varepsilon''\rangle \Downarrow \langle q;\varepsilon'''\rangle}{\langle t\, t';\varepsilon\rangle \Downarrow \langle q;\varepsilon'''\rangle} \qquad \frac{\langle t;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle}{\left\langle \mathsf{cons}_{\mu\alpha.F}\, t;\varepsilon\right\rangle \Downarrow \left\langle \mathsf{cons}_{\mu\alpha.F}\, p;\varepsilon'\right\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \left\langle \mathsf{cons}_{\mu\alpha.F}\, p;\varepsilon'\right\rangle \quad \left\langle \mathsf{fmap}_F\, (\lambda y.\, (y, \mathsf{rec}\,(x.s,y)))\, p;\varepsilon'\right\rangle \Downarrow \langle q;\varepsilon''\rangle \quad \langle s[q/x];\varepsilon''\rangle \Downarrow \langle q';\varepsilon'''\rangle}{\langle \mathsf{rec}\,(x.s,t);\varepsilon\rangle \Downarrow \langle q';\varepsilon'''\rangle}$$

$$\frac{\langle s;\varepsilon\rangle \Downarrow \langle \mathsf{delay}\, s';\varepsilon'\rangle \quad \langle t;\varepsilon'\rangle \Downarrow \langle \mathsf{delay}\, t';\varepsilon''\rangle}{\langle s \circledast t;\varepsilon\rangle \Downarrow \langle \mathsf{delay}\, (s'\ t');\varepsilon''\rangle} \qquad \frac{\langle s;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle \quad \langle t;\varepsilon'\rangle \Downarrow \langle q;\varepsilon''\rangle}{\langle s \circledcirc t;\varepsilon\rangle \Downarrow \langle p \circledcirc q;\varepsilon''\rangle} \qquad \frac{\langle t;\varepsilon\rangle \Downarrow \langle \kappa;\varepsilon'\rangle}{\langle \mathsf{wait}\, t;\varepsilon\rangle \Downarrow \langle \mathsf{wait}\, \kappa;\varepsilon'\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \langle l;\varepsilon'\rangle}{\langle \mathsf{watch}\, t;\varepsilon\rangle \Downarrow \langle \mathsf{watch}\, l;\varepsilon'\rangle} \qquad \frac{\langle s;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle \quad \langle t;\varepsilon'\rangle \Downarrow \langle q;\varepsilon''\rangle}{\langle \mathsf{select}\, s\, t;\varepsilon\rangle \Downarrow \langle \mathsf{select}\, p\, q;\varepsilon''\rangle} \qquad \frac{\langle t;\varepsilon\rangle \Downarrow \langle l;\varepsilon'\rangle}{\langle \mathsf{tail}\, t;\varepsilon\rangle \Downarrow \langle \mathsf{tail}\, l;\varepsilon'\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \langle l;\eta_N\checkmark\eta_E/\Delta\rangle \quad \eta_N(l) = p\,\langle U\rangle\, q}{\langle \mathsf{head}\, t;\varepsilon\rangle \Downarrow \langle p;\eta_N\checkmark\eta_E/\Delta\rangle} \qquad \frac{\kappa = \mathsf{alloc}\,(\Delta)}{\langle \mathsf{chan}_A;\eta_N\checkmark\eta_E/\Delta\rangle \Downarrow \langle \kappa;\eta_N\checkmark\eta_E/\Delta, \kappa : \mathsf{Chan}\, A\rangle}$$

$$\frac{\langle s;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle \quad \langle t;\varepsilon'\rangle \Downarrow \langle q;\eta_N\checkmark\eta_E/\Delta\rangle \quad l = \mathsf{alloc}\,(\eta_N\checkmark\eta_E)}{\langle s ::_A t;\varepsilon\rangle \Downarrow \langle l;\eta_N, l : \mathsf{Sig}\, A \mapsto p\,\langle\bot\rangle\, q\checkmark\eta_E/\Delta\rangle} \qquad \frac{\langle t[\mathsf{delay}\,(\mathsf{fix}\, x.t)/x];\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle}{\langle \mathsf{fix}\, x.t;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle}$$

The term $\mathsf{fmap}_F\ \vec{f} : F[\vec{A}/\vec{\alpha}] \to F[\vec{B}/\vec{\alpha}]$ takes n functions $\vec{f} = f_1, \ldots, f_n$ where each $f_i : A_i \to B_i$:

$$\begin{aligned}
&\mathsf{fmap}_{\alpha_i} \quad \vec{f}\, x = f_i\ x && \mathsf{fmap}_C \quad \vec{f}\, x = x \quad \text{if } C \text{ is of the form } 1, \ominus D, \oslash D, \mathsf{Chan}\, D\\
&\mathsf{fmap}_{C\to F}\, \vec{f}\, x = \lambda y.\, \mathsf{fmap}_F\, \vec{f}\, (x\ y) && \mathsf{fmap}_{F\times G} \quad \vec{f}\, x = \left(\mathsf{fmap}_F\, \vec{f}\, (\pi_1\, x), \mathsf{fmap}_G\, \vec{f}\, (\pi_2\, x)\right)\\
&\mathsf{fmap}_{\mathsf{Sig}\, F}\ \vec{f}\, x = \mathit{map}\, (\mathsf{fmap}_F\, \vec{f})\, x && \mathsf{fmap}_{F+G} \quad \vec{f}\, x = \mathsf{case}\, x\, \mathsf{of}\, \mathsf{in}_1\, y.\mathsf{in}_1(\mathsf{fmap}_F\, \vec{f}\, y); \mathsf{in}_2\, z.\mathsf{in}_2(\mathsf{fmap}_G\, \vec{f}\, z)\\
& && \mathsf{fmap}_{\mu\alpha_{n+1}.F}\, \vec{f}\, x = \mathsf{rec}\,(z.\mathsf{cons}_{\mu\alpha_{n+1}.F[\vec{B}/\vec{\alpha}]}\, (\mathsf{fmap}_F\, \vec{f}\, (\lambda x.\pi_2\, x)\, z), x).
\end{aligned}$$

Fig. 7. Evaluation semantics.

to evaluate a term t to a value p. To do so, the machine needs to keep track of which part of the heap has already been updated and which part still needs updating. Hence, when evaluating a term, the machine uses a *store* σ that consists of two heaps: a *now heap* η_N that stores signals that have already been updated and an *earlier heap* η_E that still requires updating. We write $\eta_N\checkmark\eta_E$ to denote a store with *now* heap η_N and *earlier* heap η_E. The *evaluation semantics*, denoted $\langle t;\varepsilon\rangle \Downarrow \langle p;\varepsilon'\rangle$ and defined in Fig. 7, describes how the machine evaluates a term t to a value p. This evaluation semantics uses an *environment* ε that consists of a store σ and a channel context Δ, written σ/Δ.

A *heap* is a sequence of assignments of the form $l : \mathsf{Sig}\, A \mapsto p\,\langle U\rangle\, q$, each of which maps a *heap location* l of type $\mathsf{Sig}\ A$ to a *stored signal* $p\,\langle U\rangle\, q$ consisting of a head value p of type A and a tail value q of type $\ominus(\mathsf{Sig}\, A)$. In addition, the stored signal contains a flag U indicating whether the signal has been updated in the current $\overset{e}{\Longrightarrow}$ step. This flag U has no bearing on the evaluation

semantics, but it is important for the step semantics of the machine discussed in section 4.2. We write $\mathrm{dom}\,(\eta)$ for the domain of a heap η, i.e. the set of heap locations stored in η; and we write $\mathrm{dom}\,(\sigma)$ for the domain of a store σ, i.e. $\mathrm{dom}\,(\eta_N \checkmark \eta_E) = \mathrm{dom}\,(\eta_N) \cup \mathrm{dom}\,(\eta_E)$.

Intuitively speaking, signals stored in the *now* heap η_N are up to date and can be safely dereferenced, whereas signals in the *earlier* heap η_E are stale and need updating before they can be dereferenced. The machine maintains the invariant that, for each store $\eta_N \checkmark \eta_E$, we have $\mathrm{dom}\,(\eta_N) \cap \mathrm{dom}\,(\eta_E) = \emptyset$, i.e. for each heap location l, there is at most one mapping for l in $\eta_N \checkmark \eta_E$. In order to allocate fresh channels and heap locations, we assume a function $\mathrm{alloc}\,(\cdot)$ so that $\mathrm{alloc}\,(\Delta)$ produces a fresh $\kappa \notin \mathrm{dom}\,(\Delta)$, and $\mathrm{alloc}\,(\sigma)$ produces a fresh $l \notin \mathrm{dom}\,(\sigma)$.

The evaluation semantics of the lambda calculus fragment of Rizzo is the standard call-by-value semantics and makes up the top half of the rules in Fig. 7. This includes the semantics for product, sum, function, and recursive types. The semantics of the primitive recursion combinator rec on recursive types $\mu\alpha.F$ is defined using a term $\mathsf{fmap}_F : (A \to B) \to F[A/\alpha] \to F[B/\alpha]$ where $\alpha \vdash F : \mathsf{type}$. Since we allow nested recursive types, fmap_F has to be defined more generally for $\alpha_1, \ldots, \alpha_n \vdash F : \mathsf{type}$ as shown at the bottom of Fig. 7.

The primitives that produce values in the $\Diamond$ modality – i.e. $\triangleright$, wait, watch, select, and tail – behave similarly to strict constructors of an algebraic data type: They eagerly evaluate their arguments to values and produce values of the form $p \triangleright q$, wait κ, watch l, select $p\ q$, and tail l, respectively. Such values of type $\Diamond A$ are then later evaluated to values of type A by the *advance semantics* of the machine presented in section 4.2. Note in particular that the evaluation semantics does *not* evaluate tail l further by looking up the tail of l stored in the *now* heap. Instead, the semantics relies on the fact that the machine's advance semantics will evaluate tail l to l once the signal at l itself has already been updated to its new value by the *update semantics* of the machine (also presented in section 4.2). This ensures that the delayed computation stored in the tail of a signal is only performed once and at the appropriate time, namely when its clock has ticked.

By contrast, head t does indeed look up the current value of the signal referred to by t and returns it. The signal constructor $::_A$ simply evaluates its arguments and stores them as a stored signal at a freshly allocated location on the *now* heap; and chan_A returns a freshly allocated channel of type $\mathsf{Chan}\ A$. Finally, fix has the standard guarded fixed point semantics: $\mathsf{fix}\ x.t$ evaluates t with x replaced by $\mathsf{delay}\ (\mathsf{fix}\ x.t)$. The latter is a value and thus does not evaluate further itself, which (in concert with the type system) ensures termination of the evaluation semantics.

4.2 Step Semantics of the Machine and Reactive Evaluation Semantics

The step semantics $\overset{e}{\Longrightarrow}$, defined in Fig. 8, describes how the machine reacts to each event e. To this end, the step semantics uses the evaluation semantics from section 4.1 as well as two additional components: The *advance semantics*, denoted $\langle p; \varepsilon\rangle \searrow^e \langle q; \varepsilon'\rangle$, describes how a value p of type $\Diamond A$ is advanced to a value q of type A given that the event e has caused the clock of p to tick. The advance semantics is used by the *update semantics*, denoted $\langle \varepsilon\rangle \overset{e}{\Longmapsto} \langle \varepsilon'\rangle$, to update the left-most signal in the *earlier* heap of the environment ε, so that it can be moved to the *now* heap, which results in the new environment ε'. Finally, the *step semantics* describes a complete computation step that is performed in reaction to an event, denoted $\langle p; \eta / \Delta\rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta'\rangle$, as well as the very first computation step that initialises a program, denoted $\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta / \Delta'\rangle$.

We first look at the step semantics: Given a well-typed term $\vdash_\Delta t : A$, the step $\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta / \Delta'\rangle$ initialises the program by evaluating t in the context of Δ to a value p using the evaluation semantics from Fig. 7. Evaluating t may cause signals to be stored in the *now* heap and additional channels to be allocated, resulting in a heap η and a new channel context Δ'.

Advance semantics $\langle p;\varepsilon\rangle \searrow^{e} \langle q;\varepsilon'\rangle$

$$\frac{\langle p_1;\varepsilon\rangle \searrow^{e} \langle p_2;\varepsilon'\rangle \qquad \langle t\ p_2;\varepsilon'\rangle \Downarrow \langle q;\varepsilon''\rangle}{\langle \mathsf{delay}\ t \ominus p_1;\varepsilon\rangle \searrow^{e} \langle q;\varepsilon''\rangle} \qquad \frac{\eta_N(l) = \mathsf{in}_1\ p_1\ \langle\top\rangle\ p_2}{\langle \mathsf{watch}\ l;\eta_N\checkmark\eta_E/\Delta\rangle \searrow^{e} \langle p_1;\eta_N\checkmark\eta_E/\Delta\rangle}$$

$$\frac{\langle t;\varepsilon\rangle \Downarrow \langle q;\varepsilon'\rangle}{\langle \mathsf{wait}\ \kappa;\varepsilon\rangle \searrow^{\kappa\mapsto t} \langle q;\varepsilon'\rangle} \qquad \frac{\langle p_i;\varepsilon\rangle \searrow^{\kappa\mapsto t} \langle q;\varepsilon'\rangle \qquad \mathsf{ticked}^{\kappa}_{\varepsilon}(p_i) \qquad \neg\mathsf{ticked}^{\kappa}_{\varepsilon}(p_{3-i})}{\langle \mathsf{select}\ p_1\ p_2;\varepsilon\rangle \searrow^{\kappa\mapsto t} \langle \mathsf{in}_1\ (\mathsf{in}_i\ q);\varepsilon'\rangle}$$

$$\frac{}{\langle \mathsf{tail}\ l;\varepsilon\rangle \searrow^{e} \langle l;\varepsilon\rangle} \qquad \frac{\langle p_1;\varepsilon\rangle \searrow^{\kappa\mapsto t} \langle q_1;\varepsilon'\rangle \qquad \langle p_2;\varepsilon'\rangle \searrow^{\kappa\mapsto t} \langle q_2;\varepsilon''\rangle \qquad \mathsf{ticked}^{\kappa}_{\varepsilon}(p_1) \qquad \mathsf{ticked}^{\kappa}_{\varepsilon}(p_2)}{\langle \mathsf{select}\ p_1\ p_2;\varepsilon\rangle \searrow^{\kappa\mapsto t} \langle \mathsf{in}_2\ (q_1,q_2);\varepsilon''\rangle}$$

Update semantics $\langle\varepsilon\rangle \overset{e}{\Longmapsto} \langle\varepsilon'\rangle$

$$\frac{\neg\mathsf{ticked}^{\kappa}_{\eta_N}(p_2)}{\langle \eta_N\checkmark l : \mathsf{Sig}\ A \mapsto p_1\ \langle U\rangle\ p_2, \eta_E/\Delta\rangle \overset{\kappa\mapsto t}{\Longmapsto} \langle \eta_N, l : \mathsf{Sig}\ A \mapsto p_1\ \langle\bot\rangle\ p_2\checkmark\eta_E/\Delta\rangle}$$

$$\frac{\mathsf{ticked}^{\kappa}_{\eta_N}(p_2) \qquad \langle p_2;\eta_N\checkmark l : \mathsf{Sig}\ A \mapsto p_1\ \langle U\rangle\ p_2, \eta_E/\Delta\rangle \searrow^{\kappa\mapsto t} \langle l';\eta'_N\checkmark l : \mathsf{Sig}\ A \mapsto p_1\ \langle U\rangle\ p_2, \eta_E/\Delta'\rangle \qquad \eta'_N(l') = q_1\ \langle V\rangle\ q_2}{\langle \eta_N\checkmark l : \mathsf{Sig}\ A \mapsto p_1\ \langle U\rangle\ p_2, \eta_E/\Delta\rangle \overset{\kappa\mapsto t}{\Longmapsto} \langle \eta'_N, l : \mathsf{Sig}\ A \mapsto q_1\ \langle\top\rangle\ q_2\checkmark\eta_E/\Delta'\rangle}$$

Step semantics $\overset{e}{\Longrightarrow}$ and $\overset{\mathsf{init}}{\Longrightarrow}$

$$\frac{\langle\cdot\checkmark\eta/\Delta\rangle \overset{e}{\Longmapsto}{}^{*} \langle\eta'\checkmark\cdot/\Delta'\rangle}{\langle p;\eta/\Delta\rangle \overset{e}{\Longrightarrow} \langle p;\eta'/\Delta'\rangle} \qquad \frac{\langle t;\cdot\checkmark\cdot/\Delta\rangle \Downarrow \langle p;\eta\checkmark\cdot/\Delta'\rangle}{\langle t;\Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p;\eta/\Delta'\rangle}$$

Reactive evaluation semantics $\langle t;\Delta\rangle \rightsquigarrow_{\tau} \langle v;\Delta'\rangle$ Events $\vdash_{\Delta} e : \mathsf{event}$

$$\frac{\vdash_{\Delta_i} e_i : \mathsf{event} \quad \text{for all } 0 \le i < n \qquad \langle t;\Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p;\eta_0/\Delta_0\rangle \overset{e_0}{\Longrightarrow} \langle p;\eta_1/\Delta_1\rangle \overset{e_1}{\Longrightarrow} \quad \ldots \quad \overset{e_{n-1}}{\Longrightarrow} \langle p;\eta_n/\Delta_n\rangle}{\langle t;\Delta\rangle \rightsquigarrow_{e_0,\ldots,e_{n-1}} \langle p[\eta_n];\Delta_n\rangle} \qquad \frac{\kappa : \mathsf{Chan}\ A \in \Delta \qquad \vdash_{\Delta} t : A}{\vdash_{\Delta} \kappa \mapsto t : \mathsf{event}}$$

where $s[\eta]$ is defined by $s[\cdot] = s$ and $s[\eta, l : \mathsf{Sig}\ A \mapsto p\ \langle U\rangle\ q] = ((s[q/\mathsf{tail}\ l])[p ::_A q/l])[\eta]$

Ticked predicate $\mathsf{ticked}^{\kappa}_{\eta}(p)$

$$\begin{aligned}
\mathsf{ticked}^{\kappa}_{\eta}(\mathsf{never}) &\Leftrightarrow \bot & \mathsf{ticked}^{\kappa}_{\eta}(\mathsf{watch}\ l) &\Leftrightarrow \exists p, q.\eta(l) = \mathsf{in}_1\ p\ \langle\top\rangle\ q\\
\mathsf{ticked}^{\kappa}_{\eta}(p \ominus q) &\Leftrightarrow \mathsf{ticked}^{\kappa}_{\eta}(q) & \mathsf{ticked}^{\kappa}_{\eta}(\mathsf{tail}\ l) &\Leftrightarrow \exists p, q.\eta(l) = p\ \langle\top\rangle\ q\\
\mathsf{ticked}^{\kappa}_{\eta}(\mathsf{wait}\ \kappa') &\Leftrightarrow \kappa = \kappa' & \mathsf{ticked}^{\kappa}_{\eta}(\mathsf{select}\ p\ q) &\Leftrightarrow \mathsf{ticked}^{\kappa}_{\eta}(p) \vee \mathsf{ticked}^{\kappa}_{\eta}(q)\\
\mathsf{ticked}^{\kappa}_{\eta_N\checkmark\eta_E/\Delta}(p) &\Leftrightarrow \mathsf{ticked}^{\kappa}_{\eta_N\checkmark\eta_E}(p) & \mathsf{ticked}^{\kappa}_{\eta_N\checkmark\eta_E}(p) &\Leftrightarrow \mathsf{ticked}^{\kappa}_{\eta_N}(p)
\end{aligned}$$

Fig. 8. Reactive evaluation semantics and step semantics of the machine.

After this initialisation, the state of the machine is represented by a tuple $\langle p; \eta / \Delta \rangle$ consisting of a value p, a heap η, and a channel context Δ. Throughout the runtime of the machine, p will not change, but p may refer to signals stored in η that *are* subject to change. Each occurrence of an event e may cause some signals stored in η to be updated. More precisely, for each stored signal $l : \mathsf{Sig}\, A \mapsto p_1 \langle U \rangle p_2$ in η, the clock of p_2 may tick in reaction to e. In that case, the signal at l must be updated with the new value of type $\mathsf{Sig}\, A$ produced by p_2 using the advance semantics. To update all such stored signals in a systematic manner, the step semantics takes the current heap η and designates it as the *earlier* heap. This means that all signals in η are considered stale and possibly need updating. The actual update process is performed – one signal at a time – by the update semantics $\overset{e}{\Longmapsto}$, which takes the left-most signal from the *earlier* heap and either moves it to the *now* heap unchanged, if the clock of the signal's tail did not tick, or it performs an update and then moves the updated signal to the *now* heap, if the clock did tick. Whether an event $\kappa \mapsto t$ causes the clock of a delayed computation p to tick is indicated by the $\mathsf{ticked}^{\kappa}_{\eta}(p)$ predicate defined at the bottom of Fig. 8. Since p may contain references to a heap η, it is included in the predicate.

To precisely relate $\mathsf{ticked}^{\kappa}_{\eta}(p)$ to the clock $\mathsf{cl}(v)$ of a value v, as defined in section 2.2, we define the clock $\mathsf{cl}_{\eta}(p)$ of a machine value p such that it satisfies the correspondence $(\mathsf{cl}_{\eta}(p))[\eta] = \mathsf{cl}(p[\eta])$:

$$\mathsf{cl}_{\eta}(\mathsf{never}) = \emptyset \qquad \mathsf{cl}_{\eta}(\mathsf{wait}\,\kappa) = \{\kappa\}$$
$$\mathsf{cl}_{\eta}(\mathsf{watch}\, l) = \{l\} \qquad \mathsf{cl}_{\eta}(\mathsf{select}\ p\ q) = \mathsf{cl}_{\eta}(p) \cup \mathsf{cl}_{\eta}(q)$$
$$\mathsf{cl}_{\eta}(p \circledcirc q) = \mathsf{cl}_{\eta}(q) \qquad \mathsf{cl}_{\eta}(\mathsf{tail}\, l) = \mathsf{cl}_{\eta_1}(q) \quad \text{if } \eta = \eta_1, l : \mathsf{Sig}\, A \mapsto p \langle U \rangle q, \eta_2$$

During the execution of a reactive step $\langle p; \eta / \Delta \rangle \overset{\kappa \mapsto t}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$, the machine maintains the following invariant for any environment $\eta_N \checkmark \eta_E / \Delta''$ throughout the update process:

$$\mathsf{ticked}^{\kappa}_{\eta_N}(q) \text{ iff } \kappa \in \mathsf{cl}_{\eta}(q) \text{ or there is some } l \in \mathsf{cl}_{\eta}(q) \text{ with } \eta_N(l) = \mathsf{in}_1\, q_1 \langle \top \rangle q_2. \tag{1}$$

That is, the clock of a delayed computation q ticks in response to an event on channel κ (and therefore can be advanced) iff the clock contains κ or a reference l to a partial signal that has produced a fresh value during this reactive step. Note that the relevant clock $\mathsf{cl}_{\eta}(q)$ is with respect to η, i.e. the heap from *before* the start of the reactive step, because the timing information in η_N is for the *next* step. We state and prove the above observations more formally in section 5.4.

Whenever the clock $\mathsf{cl}_{\eta}(p_2)$ of a stored signal $p_1 \langle U \rangle p_2$ ticks, the tail p_2 is advanced to produce the new state of the signal. In general, the advance semantics takes a delayed computation $p : \circledcirc A$ whose clock has ticked, i.e. $\mathsf{ticked}^{\kappa}_{\eta_N}(p)$, and performs the delayed computation to obtain a value q of type A, denoted $\langle p; \eta_N \checkmark \eta_E / \Delta \rangle \searrow^{\kappa \mapsto t} \langle q; \eta'_N \checkmark \eta_E / \Delta' \rangle$. Similarly to the evaluation semantics, the advance semantics may allocate new signals on the *now* heap and create fresh channels, but it leaves the *earlier* heap unchanged. In the cases for $p = \mathsf{watch}\ l$ and $p = \mathsf{wait}\ \kappa$, the advance semantics can simply look up the desired terms in η_N and $\kappa \mapsto t$, respectively, because we know that $\mathsf{ticked}^{\kappa}_{\eta_N}(p)$ holds. In the case of $p = \mathsf{select}\ p_1\ p_2$, the advance semantics consults $\mathsf{ticked}^{\kappa}_{\eta_N}(\cdot)$ to decide which of the two delayed computations to perform. Since $\mathsf{ticked}^{\kappa}_{\eta_N}(p)$ holds, we know that at least one of the two has ticked. In the case of $p = \mathsf{delay}\ t \circledcirc p_1$, the advance semantics performs the delayed computation p_1 to obtain a value p_2 and then evaluates the function application $t\ p_2$. Finally, in the case of $p = \mathsf{tail}\ l$, the advance semantics simply produces l, because it can rely on the fact that l points into the *now* heap, which the update semantics has already updated.

We define the reactive evaluation semantics $\langle t; \Delta \rangle \rightsquigarrow_{\tau} \langle v; \Delta_n \rangle$ in Fig. 8 as the result of the machine reacting to the events in τ where each event e_i in τ is well-typed according to the judgement $\vdash_{\Delta_i} e_i : \mathsf{event}$ also defined in Fig. 8. The resulting machine value p together with the heap η_n represent the value $v = p[\eta_n]$, which we obtain by taking each stored signal $l : \mathsf{Sig}\, A \mapsto p_1 \langle U \rangle p_2$ in η_n and replacing $\mathsf{tail}\, l$ with p_2 and l with $p_1 ::_A p_2$ in p.

4.3 Practical Considerations

While the operational semantics provides a precise account of the operational behaviour of Rizzo and thus allows us to give formal statements of the operational guarantees provided by the type system, it does not try to give an *efficient* implementation strategy. How to efficiently implement algebraic concepts like variable substitution and recursive data types is well studied in the literature. In the following, we focus on how the semantics of signals may be implemented efficiently.

Instead of two separate heaps to store signals, an implementation could use a single, global heap for *now* signals, *earlier* signals, and any other data types that need heap allocation such as product, sum and recursive types. The *now* and *earlier* heaps are instead jointly represented by a linked list data structure by making each signal store a reference to its two neighbours. The ✓ divider between the *now* and *earlier* heap is then represented by a reference into the linked list.

In a practical implementation, we have some memory management strategy that deallocates unused data from the global heap – including stored signals. However, signals stored on the heap take up space *and time*, because each stored signal is checked by the update semantics and updated if necessary. It is thus important that unused memory is freed in a timely fashion so that as few unused signals as possible are updated. That makes *automatic reference counting*, which has seen renewed attention for efficient implementations of functional languages [Lorenzen and Leijen 2022; Reinking et al. 2021; Ullrich and de Moura 2021], an ideal memory management strategy for Rizzo. It frees unused memory immediately, and thus no unused signals are ever updated. Moreover, since the type system rules out circularity (cf. Theorem 4.1 in section 4.6), programs cannot produce circular reference chains, which automatic reference counting would not be able to detect.

Finally, there are further small optimisations that a practical implementation could perform in order to reduce the number of allocations of stored signals. For example, when the update semantics updates a signal at location l, it advances the tail of that signal. This will produce a new signal at l', whose data is copied into l. An efficient implementation can avoid the additional allocation of l' and the subsequent copying by simply writing the new signal directly into l to begin with. This is safe, because we can rule out that the process of advancing the tail of l will itself read from l, due to the simple fact that l resides in the *earlier* heap.

4.4 Example Run of the Machine

To illustrate the machine, we show how it executes a small example program involving *sample*. This and two more examples are developed in more detail in Appendix A and the accompanying Lean formalisation (both found in the supplementary material).

Our example program t assumes a channel context $\Delta = \{\kappa_1 : \mathsf{Chan}\ \mathsf{Nat}, \kappa_2 : \mathsf{Chan}\ \mathit{Char}\}$:

$$t = \mathbf{let}\ xs = 0 :: \mathit{mkSig}\ (\mathsf{wait}\ \kappa_1)\ \mathbf{in}\ \mathbf{let}\ ys = \texttt{'a'} :: \mathit{mkSig}\ (\mathsf{wait}\ \kappa_2)\ \mathbf{in}\ \mathit{sample}\ xs\ ys$$

This program first constructs two signals xs and ys from the two channels κ_1 and κ_2 and then samples from ys using xs. To avoid clutter, we elide heap locations that are not referenced anywhere, leave out the type annotations of all heap locations, and do not spell out the values p_i:

$$\begin{aligned}
\langle t; \Delta\rangle &\overset{\text{init}}{\Longrightarrow} \langle l_3; l_1 \mapsto 0\,\langle\bot\rangle\, p_1, l_2 \mapsto \texttt{'a'}\,\langle\bot\rangle\, p_2, l_3 \mapsto (0, \texttt{'a'})\,\langle\bot\rangle\, p_3 / \Delta\rangle \\
&\overset{\kappa_1 \mapsto 1}{\Longrightarrow} \langle l_3; l_1 \mapsto 1\,\langle\top\rangle\, p_1, l_2 \mapsto \texttt{'a'}\,\langle\bot\rangle\, p_2, l_3 \mapsto (1, \texttt{'a'})\,\langle\top\rangle\, p_3 / \Delta\rangle \\
&\overset{\kappa_2 \mapsto \texttt{'b'}}{\Longrightarrow} \langle l_3; l_1 \mapsto 1\,\langle\bot\rangle\, p_1, l_2 \mapsto \texttt{'b'}\,\langle\top\rangle\, p_2, l_3 \mapsto (1, \texttt{'a'})\,\langle\bot\rangle\, p_3 / \Delta\rangle \\
&\overset{\kappa_1 \mapsto 2}{\Longrightarrow} \langle l_3; l_1 \mapsto 2\,\langle\top\rangle\, p_1, l_2 \mapsto \texttt{'b'}\,\langle\bot\rangle\, p_2, l_3 \mapsto (2, \texttt{'b'})\,\langle\top\rangle\, p_3 / \Delta\rangle
\end{aligned}$$

The initialisation step allocates three signals on the heap. Each of the first two, at l_1 and l_2, updates whenever the corresponding channel κ_i receives an input since $\mathsf{cl}_\eta\,(\mathsf{tail}\ l_i) = \mathsf{cl}_\eta\,(p_i) =$

$$\frac{}{\vdash_\Delta \cdot : \mathsf{now}} \qquad \frac{\vdash_\Delta \eta : \mathsf{now} \qquad \vdash_\Delta^{|\eta|} p : A \qquad \vdash_\Delta^{|\eta|} q : \textcircled{\exists}(\mathrm{Sig} A)}{\vdash_\Delta \eta, l : \mathrm{Sig}\, A \mapsto p \,\langle U \rangle\, q : \mathsf{now}}$$

$$\frac{}{\vdash_\Delta^H \cdot : \mathsf{earlier}} \qquad \frac{\vdash_\Delta^{H, l:\mathrm{Sig}\, A} \eta : \mathsf{earlier} \qquad \vdash_\Delta^H p : A \qquad \vdash_\Delta^H q : \textcircled{\exists}(\mathrm{Sig} A)}{\vdash_\Delta^H l : \mathrm{Sig}\, A \mapsto p \,\langle U \rangle\, q, \eta : \mathsf{earlier}}$$

$$\frac{\vdash_\Delta^{|\eta|} t : A \qquad \vdash_\Delta \eta : \mathsf{now}}{\vdash_\Delta^\eta t : A} \qquad \frac{\vdash_\Delta \eta_N : \mathsf{now} \qquad \vdash_\Delta^{|\eta_N|} \eta_E : \mathsf{earlier}}{\vdash \eta_N \checkmark \eta_E / \Delta : \mathsf{env}}$$

Fig. 9. Well-typed *now* heaps ($\vdash_\Delta \eta : \mathsf{now}$), *earlier* heaps ($\vdash_\Delta^H \eta : \mathsf{earlier}$), and environments ($\vdash \varepsilon : \mathsf{env}$).

$\mathsf{cl}_\eta\,(\mathsf{wait}\,\kappa_i) = \{\kappa_i\}$. The signal at l_3, however, only updates whenever κ_1 receives an input since $\mathsf{cl}_\eta\,(\mathsf{tail}\,l_3) = \mathsf{cl}_\eta\,(p_3) = \mathsf{cl}_\eta\,(\mathsf{tail}\,l_1) = \{\kappa_1\}$. That is, given $\tau = \kappa_1 \mapsto 1, \kappa_2 \mapsto \texttt{'b'}, \kappa_1 \mapsto 2$, we have the reactive evaluation $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle (2, \texttt{'b'}) :: v; \Delta\rangle$ for some $v : \textcircled{\exists}\ (\mathrm{Sig}\ (\mathrm{Nat} \times \mathit{Char}))$.

As mentioned above, we have elided heap locations that are not referenced anywhere. For example, after the first reactive step $\overset{\kappa_1 \mapsto 1}{\Longrightarrow}$, the machine also allocates a heap location l_4 with the same value as l_1 and inserts it just to the left of l_1. Similarly, the machine allocates a location l_5 just to the left of l_3 with the same value as l_3. In a practical implementation, these intermediate heap locations can be garbage collected or avoided altogether as described in section 4.3.

4.5 Extended Type System and Type Preservation

As discussed in section 4.1, the evaluation semantics creates fresh channels in the channel context Δ and allocates new signals in the *now* heap η_N. To give a precise account of the type preservation property satisfied by Rizzo, we generalise the typing judgement $\Gamma \vdash_\Delta t : A$ to an extended typing judgement of the form $\Gamma \vdash_\Delta^H t : A$ so that it also has a *heap context* H. The latter is a sequence of heap location typings of the form $l : \mathrm{Sig}\, A$, each of which indicates that there is a signal of type $\mathrm{Sig}\, A$ at heap location l. All typing rules from Fig. 3 carry over to the extended typing judgement with the heap context H being the same in premises and conclusions (just like Δ). For example, the rule for lambda abstractions is as follows in the extended system:

$$\frac{\Gamma, x : A \vdash_\Delta^H t : B}{\Gamma \vdash_\Delta^H \lambda x.t : A \to B}$$

In addition, we have one new typing rule that simply considers heap locations from H as well-typed:

$$\frac{l : \mathrm{Sig}\, A \in H}{\Gamma \vdash_\Delta^H l : \mathrm{Sig}\, A}$$

This extended type system is only needed to formulate and to prove the operational properties of Rizzo. Programs are still expected to be typed according to the "surface" type system from Fig. 3. However, this "surface" type system is a special case of the extended type system where H is empty.

The evaluation semantics preserves typing with respect to the extended typing judgement. In order to state this precisely, we define the heap context $|\eta|$ of a heap η as the sequence of heap locations and their typing from η, i.e. $|\cdot| = \cdot$ and $|\eta, l : \mathrm{Sig}\, A \mapsto p \,\langle U \rangle\, q| = |\eta|, l : \mathrm{Sig}\, A$, where we write $\cdot$ for the empty heap.

Finally, to precisely state the operational guarantees of Rizzo, we also give typing judgements for *now* and *earlier* heaps ($\vdash_\Delta \eta : \mathsf{now}$ and $\vdash_\Delta^H \eta : \mathsf{earlier}$) as well as environments ($\vdash \varepsilon : \mathsf{env}$) in Fig. 9. In addition, we also define the shorthand $\vdash_\Delta^\eta t : A$, which states that both t and η are well-typed.

4.6 Main Metatheoretical Results

Using the operational semantics, we can now give a precise account of the operational guarantees that Rizzo's type system provides, namely productivity, causality, and the absence of space leaks.

4.6.1 Productivity. Recall the definition of the well-typed event judgement $\vdash_\Delta e : \mathsf{event}$ from Fig. 8. A *well-formed reactive sequence* is of the form:

$$\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta_0 / \Delta_0\rangle \overset{e_0}{\Longrightarrow} \langle p; \eta_1 / \Delta_1\rangle \overset{e_1}{\Longrightarrow} \ldots \overset{e_{n-1}}{\Longrightarrow} \langle p; \eta_n / \Delta_n\rangle \text{ with } \vdash_{\Delta_i} e_i : \mathsf{event} \text{ for } 0 \le i < n \quad (2)$$

Productivity states that the machine can always extend such a sequence from a well-typed term:

Theorem 4.1 (productivity). *Let $\vdash_\Delta t : A$ be a well-typed term.*

(i) There is a step $\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta_0 / \Delta_0\rangle$.
(ii) Given a well-formed reactive sequence of the form (2) and an event $\vdash_{\Delta_n} e_n : \mathsf{event}$, there is a step $\langle p; \eta_n / \Delta_n\rangle \overset{e_n}{\Longrightarrow} \langle p; \eta_{n+1} / \Delta_{n+1}\rangle$.

This means that, given a well-typed term $\vdash_\Delta t : A$ and an infinite sequence of well-typed events $(\vdash_{\Delta_i} e : \mathsf{event}_i)_{i\in\mathbb{N}}$, we obtain an infinite well-formed reactive sequence

$$\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta_0 / \Delta_0\rangle \overset{e_0}{\Longrightarrow} \langle p; \eta_1 / \Delta_1\rangle \overset{e_1}{\Longrightarrow} \ldots \quad (3)$$

Moreover, the output produced by this reactive sequence, namely the value p along with any signals it may refer to in the heaps η_i, is well-typed:

Theorem 4.2 (type preservation). *Given a well-typed term $\vdash_\Delta t : A$ and an infinite well-formed reactive sequence of the form (3), we have $\vdash^{\eta_i}_{\Delta_i} p : A$ for all $i \ge 0$.*

Here we use the judgement $\vdash^{\eta_i}_{\Delta_i} p : A$ from Fig. 9 stating that both p and η_i are well-typed.

We can state the above productivity and the type preservation properties in terms of the reactive evaluation relation $\rightsquigarrow_\tau$. The productivity property follows from Theorem 4.1 and the fact that $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle t'; \Delta'\rangle$ implies that t' is a value according to the grammar in Fig. 1:

Corollary 4.3 (productivity). *Let $\vdash_\Delta t : A$ be a well-typed term.*

(i) There is a reactive evaluation $\langle t; \Delta\rangle \rightsquigarrow . \langle v; \Delta'\rangle$, where $\cdot$ is the empty sequence of events.
(ii) If $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$ and $\vdash_{\Delta'} e : \mathsf{event}$, then there is a reactive evaluation $\langle t; \Delta\rangle \rightsquigarrow_{\tau,e} \langle w; \Delta''\rangle$.

In turn, type preservation follows from Theorem 4.2 and the fact that $\vdash^{\eta}_{\Delta} p : A$ implies $\vdash_\Delta p[\eta] : A$:

Corollary 4.4 (type preservation). *If $\vdash_\Delta t : A$ and $\langle t; \Delta\rangle \rightsquigarrow_\tau \langle v; \Delta'\rangle$, then $\vdash_{\Delta'} v : A$.*

4.6.2 Causality. A Rizzo term $\vdash_\Delta t : A$ is called *causal* if, for any infinite well-formed reactive sequence of the form (3), each state $\langle p; \eta_n / \Delta_n\rangle$ in the sequence only depends on the initial state $\langle t; \Delta\rangle$ and all events e_i with $i < n$.

Theorem 4.5 (causality). *Suppose the infinite well-formed reactive sequences (3) and*

$$\langle t; \Delta\rangle \overset{\mathsf{init}}{\Longrightarrow} \left\langle p'; \eta'_0 / \Delta'_0\right\rangle \overset{e'_0}{\Longrightarrow} \left\langle p'; \eta'_1 / \Delta'_1\right\rangle \overset{e'_1}{\Longrightarrow} \ldots$$

Let $n \in \mathbb{N}$ and suppose $e_i = e'_i$ for all $i < n$. Then $\langle p; \eta_n / \Delta_n\rangle = \left\langle p'; \eta'_n / \Delta'_n\right\rangle$.

4.6.3 No Space Leaks. The absence of space leaks is a direct consequence of the productivity property in Theorem 4.1: The operational semantics is by definition free of space leaks since after each step $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$, all old signals have been overwritten with their new value. Hence, the program cannot retain old input data. Theorem 4.1 shows that this in-place update semantics is safe, and that the program never tries to dereference old data. The old data is kept in the *earlier* heap and any attempt by the program to dereference a signal from that heap would result in a stuck execution, which Theorem 4.1 rules out.

5 Metatheory

In this section, we give an overview of how we have proved the main results presented in section 4.6 as well as the timing correspondence (1). The complete formalisation of these proofs in Lean can be found in the supplementary material to this article [Bahr 2026]. The three main theorems we prove are Theorem 4.1 (productivity), Theorem 4.2 (type preservation), and Theorem 4.5 (causality), while the absence of space leaks follows by definition of the operational semantics and Theorem 4.1.

5.1 Causality

The causality property of Theorem 4.5 is an immediate consequence of the fact that the operational semantics is deterministic in each of its components, in particular the step semantics:

Lemma 5.1 (determinism).

(i) If $\langle t; \Delta \rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p_1; \eta_1 / \Delta_1 \rangle$ *and* $\langle t; \Delta \rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p_2; \eta_2 / \Delta_2 \rangle$, *then* $\langle p_1; \eta_1 / \Delta_1 \rangle = \langle p_2; \eta_2 / \Delta_2 \rangle$.

(ii) If $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p_1; \eta_1 / \Delta_1 \rangle$ *and* $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p_2; \eta_2 / \Delta_2 \rangle$, *then* $\langle p_1; \eta_1 / \Delta_1 \rangle = \langle p_2; \eta_2 / \Delta_2 \rangle$.

This determinism property is proved by showing that all parts of the operational semantics (evaluation, advance, update, and step semantics) are deterministic using a straightforward argument by structural induction on the rules of the operational semantics.

5.2 Productivity & Type Preservation

Theorem 4.1 (productivity) and Theorem 4.2 (type preservation) are a consequence of two standard properties [Wright and Felleisen 1994]: *progress*, i.e. given a well-typed starting point, the operational semantics produces a result, and *preservation*, i.e. the produced result is itself well-typed again. However, instead of a syntactic proof, we use a logical relations argument for the progress property.

In order to make the progress and preservation properties precise, we have introduced the extended typing judgement $\Gamma \vdash^{H}_{\Delta} t : A$ in section 4.5 along with typing judgements for heaps and environments in Fig. 9. Thus, we can formally describe the type preservation properties as follows:

Proposition 5.2 (type preservation).

(i) If $\vdash^{\eta_N}_{\Delta} t : A$ *and* $\langle t; \eta_N \checkmark \eta_E / \Delta \rangle \Downarrow \langle p; \eta'_N \checkmark \eta'_E / \Delta' \rangle$, *then* $\vdash^{\eta'_N}_{\Delta'} p : A$.

(ii) If $\vdash^{\eta_N}_{\Delta} p : \Box A$, $\vdash_{\Delta} e : \mathsf{event}$, *and* $\langle p; \eta_N \checkmark \eta_E / \Delta \rangle \searrow^{e} \langle q; \eta'_N \checkmark \eta'_E / \Delta' \rangle$, *then* $\vdash^{\eta'_N}_{\Delta'} q : A$.

(iii) If $\vdash \varepsilon : \mathsf{env}$, $\vdash_{\Delta} e : \mathsf{event}$, *and* $\langle \varepsilon \rangle \overset{e}{\Longmapsto} \langle \varepsilon' \rangle$, *then* $\vdash \varepsilon' : \mathsf{env}$.

(iv) If $\vdash_{\Delta} t : A$ *and* $\langle t; \Delta \rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta / \Delta' \rangle$, *then* $\vdash^{\eta}_{\Delta'} p : A$.

(v) If $\vdash_{\Delta} \eta : \mathsf{now}$, $\vdash_{\Delta} e : \mathsf{event}$, *and* $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$, *then* $\vdash_{\Delta'} \eta' : \mathsf{now}$.

All five type preservation properties can be proved by structural induction on the definition of the operational semantics. To obtain Theorem 4.2, we need two additional properties:

Lemma 5.3 (weakening). *If* $\Gamma \vdash^{H}_{\Delta} t : A$, $H \subseteq H'$, *and* $\Delta \subseteq \Delta'$, *then* $\Gamma \vdash^{H'}_{\Delta'} t : A$.

Lemma 5.4. *If* $\vdash_{\Delta} \eta : \mathsf{now}$, $\vdash_{\Delta} e : \mathsf{event}$, *and* $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$, *then* $|\eta| \subseteq |\eta'|$ *and* $\Delta \subseteq \Delta'$.

Lemma 5.3 states the weakening of the term typing judgement and follows by an induction on the derivation of $\Gamma \vdash^{H}_{\Delta} t : A$, while Lemma 5.4 states that the reactive step semantics grows the heap context and the channel context, which follows by an induction on $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$.

Proof of Theorem 4.2 (type preservation). We prove $\vdash^{\eta_i}_{\Delta_i} p : A$ by induction on i. If $i = 0$, then $\vdash^{\eta_0}_{\Delta_0} p : A$ follows by Proposition 5.2 (iv). If $i = j + 1$, then we have $\vdash^{\eta_j}_{\Delta_j} p : A$ by induction hypothesis, which means $\vdash^{|\eta_j|}_{\Delta_j} p : A$ and $\vdash_{\Delta_j} \eta_j : \mathsf{now}$. By Lemma 5.4, we have $|\eta_j| \subseteq |\eta_i|$ and $\Delta_j \subseteq \Delta_i$. Thus $\vdash^{|\eta_i|}_{\Delta_i} p : A$ follows by Lemma 5.3, and $\vdash_{\Delta_i} \eta_i : \mathsf{now}$ follows by Proposition 5.2 (v). □

For each preservation property in Proposition 5.2, we have a corresponding progress property, which states that the computation step in the preservation property exists:

Proposition 5.5 (progress).

(i) *If* $\vdash^{\eta_N}_{\Delta} t : A$, *then there are* $p, \eta'_N, \eta'_E, \Delta'$ *with* $\langle t; \eta_N \checkmark \eta_E / \Delta \rangle \Downarrow \langle p; \eta'_N \checkmark \eta'_E / \Delta' \rangle$.

(ii) *If* $\vdash^{\eta_N}_{\Delta} p : \circledexists A$, $\vdash_{\Delta} \kappa \mapsto t : \mathsf{event}$, *and* $\mathsf{ticked}^{\kappa}_{\eta_N}(p)$, *then there are* $q, \eta'_N, \eta'_E, \Delta'$ *with* $\langle p; \eta_N \checkmark \eta_E / \Delta \rangle \searrow^{\kappa \mapsto t} \langle q; \eta'_N \checkmark \eta'_E / \Delta' \rangle$.

(iii) *If* $\vdash \eta_N \checkmark \eta_E / \Delta : \mathsf{env}$, $\vdash_{\Delta} e : \mathsf{event}$, *and* η_E *is non-empty, then there are* σ', Δ' *with* $\langle \eta_N \checkmark \eta_E / \Delta \rangle \overset{e}{\Longmapsto} \langle \sigma' / \Delta' \rangle$.

(iv) *If* $\vdash_{\Delta} t : A$, *then there are* p, η, Δ' *with* $\langle t; \Delta \rangle \overset{\mathsf{init}}{\Longrightarrow} \langle p; \eta / \Delta' \rangle$.

(v) *If* $\vdash_{\Delta} \eta : \mathsf{now}$, $\vdash_{\Delta} e : \mathsf{event}$, *then there are* η', Δ' *with* $\langle p; \eta / \Delta \rangle \overset{e}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$.

The proof of (i) uses a logical relations argument, which we sketch in section 5.3 below, whereas (ii) follows by induction on the machine value of type $\circledexists A$, appealing to (i) in the cases for wait and $\circledcirc$. The remaining properties (iii) to (v) follow from (i), (ii), and the corresponding type preservation properties in Proposition 5.2.

Proof of Theorem 4.1 (productivity). Part (i) follows immediately from Proposition 5.5 (iv). To show (ii), assume a well-formed reactive sequence of the form (2). By induction on the length n of the sequence, we can show that $\vdash_{\Delta_n} \eta_n : \mathsf{now}$ using Proposition 5.2 (iv)-(v). We can then apply Proposition 5.5 (v) to obtain the desired step $\langle p; \eta_n / \Delta_n \rangle \overset{e_n}{\Longrightarrow} \langle p; \eta_{n+1} / \Delta_{n+1} \rangle$. □

5.3 Logical Relation

We sketch the logical relations argument that underpins the proof of Proposition 5.5 (i). This proof proceeds by constructing for each closed type A and each environment ε a set of terms $\mathcal{T}[\![A]\!](\varepsilon)$ that are semantically of type A in the context of the environment ε. By construction, each term in $\mathcal{T}[\![A]\!](\varepsilon)$ evaluates to a value. Proposition 5.5 (i) then follows from the fundamental property of the logical relation $\mathcal{T}[\![A]\!](\varepsilon)$, which states that all well-typed terms of type A are in $\mathcal{T}[\![A]\!](\varepsilon)$.

The proof of the fundamental property relies on the fact that the logical relation satisfies *Kripke monotonicity*, which means that it is closed under a suitable ordering $\sqsubseteq$ on ε. This ordering captures the fact that the evaluation semantics may allocate additional signals and channels. On heaps and on channel contexts, $\sqsubseteq$ is defined as the subsequence relation. That is, $\Delta \sqsubseteq \Delta'$ iff Δ can be obtained from Δ' by removing some of its elements. On heaps, $\sqsubseteq$ is defined analogously. We lift this ordering to environments by pointwise ordering on the *now* heap and the channel context components:

$$\eta_N \checkmark \eta_E / \Delta \sqsubseteq \eta'_N \checkmark \eta'_E / \Delta' \qquad \text{iff} \qquad \eta_N \sqsubseteq \eta'_N \wedge \Delta \sqsubseteq \Delta' \wedge \eta_E = \eta'_E$$

Note that the *earlier* heap remains constant. This matches the following property of the evaluation semantics, which can be proved by a straightforward induction on the evaluation semantics:

$$
\begin{aligned}
\mathcal{V}^\rho[\![1]\!](\varepsilon) &= \{()\}, \\
\mathcal{V}^\rho[\![F \times G]\!](\varepsilon) &= \left\{(p_1, p_2) \,\middle|\, p_1 \in \mathcal{V}^\rho[\![F]\!](\varepsilon) \wedge p_2 \in \mathcal{V}^\rho[\![G]\!](\varepsilon)\right\}, \\
\mathcal{V}^\rho[\![F + G]\!](\varepsilon) &= \left\{\mathsf{in}_1\, p \,\middle|\, p \in \mathcal{V}^\rho[\![F]\!](\varepsilon)\right\} \cup \left\{\mathsf{in}_2\, p \,\middle|\, p \in \mathcal{V}^\rho[\![G]\!](\varepsilon)\right\} \\
\mathcal{V}^\rho[\![\textcircled{\forall} A]\!](\varepsilon) &= \{p \mid \varepsilon \Vdash p : \textcircled{\forall} A\} \qquad \text{where} \quad \eta_N \checkmark \eta_E / \Delta \Vdash t : A \quad \text{iff} \quad \vdash_\Delta^{|\eta_N|} t : A \\
\mathcal{V}^\rho[\![\textcircled{\exists} A]\!](\varepsilon) &= \{p \mid \varepsilon \Vdash p : \textcircled{\exists} A\} \\
\mathcal{V}^\rho[\![A \to F]\!](\varepsilon) &= \left\{\lambda x.t \,\middle|\, \varepsilon \Vdash \lambda x.t : A \to F[\rho] \wedge \forall \varepsilon' \sqsupseteq \varepsilon. p \in \mathcal{V}[\![A]\!](\varepsilon'). t[p/x] \in \mathcal{T}^\rho[\![F]\!](\varepsilon')\right\} \\
\mathcal{V}^\rho[\![\mathsf{Chan}\, A]\!](\sigma/\Delta) &= \{\kappa \mid \kappa : \mathsf{Chan}\, A \in \Delta\} \\
\mathcal{V}^\rho[\![\mathsf{Sig}\, F]\!](\eta_N \checkmark \eta_E / \Delta) &= \left\{l \,\middle|\, l : \mathsf{Sig}\, F[\rho] \mapsto p\, \langle U \rangle\, q \in \eta_N, p \in \mathcal{V}^\rho[\![F]\!](\eta_N \checkmark \eta_E / \Delta)\right\} \\
\mathcal{V}^\rho[\![\mu\alpha.F]\!](\varepsilon) &= \mathit{lfp}(\Psi_\rho^{\mu\alpha.F})(\varepsilon), \quad \text{where } \mathit{lfp}(\Psi) = \bigcap \{X \mid X \text{ is Kripke monotone and } \Psi(X) \subseteq X\} \\
&\qquad \Psi_\rho^{\mu\alpha.F}(X)(\varepsilon) = \left\{\mathsf{cons}_{(\mu\alpha.F)[\rho]}\, p \,\middle|\, p \in \mathcal{V}^{\rho[\alpha \mapsto (X, (\mu\alpha.F)[\rho])]}[\![F]\!](\varepsilon)\right\} \\
\mathcal{V}^\rho[\![\alpha]\!](\varepsilon) &= \{p \in X(\varepsilon) \mid \varepsilon \Vdash p : A\} \quad \text{if } \rho(\alpha) = (X, A)
\end{aligned}
$$

$$
\mathcal{V}[\![A]\!](\varepsilon) = \mathcal{V}^\emptyset[\![A]\!](\varepsilon) \qquad \mathcal{T}^\rho[\![F]\!](\varepsilon) = \left\{t \,\middle|\, \exists p, \varepsilon'. \langle t; \varepsilon\rangle \Downarrow \langle p; \varepsilon'\rangle \wedge p \in \mathcal{V}^\rho[\![F]\!](\varepsilon')\right\}
$$

$$
\mathcal{H}(\varepsilon) = \left\{\eta \,\middle|\, \forall l : \mathsf{Sig}\, A \mapsto p\, \langle U \rangle\, q \in \eta. p \in \mathcal{V}[\![A]\!](\varepsilon)\right\}
$$

$$
C[\![\cdot]\!](\varepsilon) = \{*\} \qquad C[\![\Gamma, x : A]\!](\varepsilon) = \left\{\gamma[x \mapsto p] \,\middle|\, \gamma \in C[\![\Gamma]\!](\varepsilon), p \in \mathcal{V}[\![A]\!](\varepsilon)\right\}
$$

Fig. 10. Logical relation. Recall that A, B range over closed types, and F, G range over possibly open types.

Lemma 5.6. *If* $\langle t; \varepsilon\rangle \Downarrow \langle p; \varepsilon'\rangle$, *then* $\varepsilon \sqsubseteq \varepsilon'$.

Fig. 10 defines the logical relations used for proving the progress property. In addition to the term relation $\mathcal{T}^\rho[\![F]\!](\varepsilon)$, we also define a corresponding value relation $\mathcal{V}^\rho[\![F]\!](\varepsilon)$. Both $\mathcal{T}^\rho[\![F]\!](\varepsilon)$ and $\mathcal{V}^\rho[\![F]\!](\varepsilon)$ are defined more generally for open types F, i.e. F may contain free type variables. To account for this, both logical relations have the additional component ρ, which is a finite map from type variables to pairs (X, A) consisting of a semantic type X and a closed type A. More precisely, X is a function from environments to sets of values of type A. In the definition of the value relation, we use the notation $F[\rho]$ to apply the syntactic substitution ρ^{syn} to a type F, where $\rho^{\mathsf{syn}}(\alpha) = A$ iff $\rho(\alpha) = (X, A)$. The definition of $\mathcal{V}^\rho[\![\mu\alpha.F]\!](\varepsilon)$ uses a standard least pre-fixed point construction on semantic types, denoted by *lfp*, on a function $\Psi_\rho^{\mu\alpha.F}$. To simplify the proof, *lfp* is limited to semantic types X that are Kripke monotone, i.e. $X(\varepsilon) \subseteq X(\varepsilon')$ whenever $\varepsilon \sqsubseteq \varepsilon'$. The least pre-fixed point is well-defined since $\Psi_\rho^{\mu\alpha.F}$ is monotone, i.e. $\Psi_\rho^{\mu\alpha.F}(X)(\varepsilon) \subseteq \Psi_\rho^{\mu\alpha.F}(Y)(\varepsilon)$ whenever $X(\varepsilon) \subseteq Y(\varepsilon)$.

Since the proof of the fundamental property proceeds by induction on the typing judgement, we also have to consider open terms. To this end, we also define a corresponding context relation $C[\![\Gamma]\!](\varepsilon)$, which contains term substitutions γ such that $\gamma(x) \in \mathcal{V}[\![A]\!](\varepsilon)$ iff $x : A \in \Gamma$; and a heap relation $\mathcal{H}(\varepsilon)$ to capture semantically well-typed *now* heaps with respect to an environment ε.

Proposition 5.7 (fundamental property). *If* $\Gamma \vdash_\Delta^{|\eta_N|} t : A$, $\eta_N \checkmark \eta_E / \Delta \sqsubseteq \varepsilon$, $\eta_N \in \mathcal{H}(\varepsilon)$, *and* $\gamma \in C[\![\Gamma]\!](\varepsilon)$, *then* $t\gamma \in \mathcal{T}[\![A]\!](\varepsilon)$.

The fundamental property is proved by a lengthy induction on $\Gamma \vdash_\Delta^{|\eta_N|} t : A$. The proof relies on the fact that the value, context, and heap relations are Kripke monotone:

Lemma 5.8. *Let* $\varepsilon \sqsubseteq \varepsilon'$. *Then* $\mathcal{V}[\![A]\!](\varepsilon) \subseteq \mathcal{V}[\![A]\!](\varepsilon')$, $C[\![\Gamma]\!](\varepsilon) \subseteq C[\![\Gamma]\!](\varepsilon')$, *and* $\mathcal{H}(\varepsilon) \subseteq \mathcal{H}(\varepsilon')$

Proposition 5.7 is much more general than what we need to prove Proposition 5.5 (i), so that the induction hypothesis is strong enough for the induction argument to succeed. Proposition 5.5 (i) follows immediately from the following instantiation of Proposition 5.7 to closed terms:

Corollary 5.9 (fundamental property). *If* $\vdash_{\Delta}^{\eta_N} t : A$, *then* $t \in \mathcal{T}[\![A]\!](\eta_N \checkmark \eta_E / \Delta)$.

5.4 Timing of Delayed Computations

Finally, we make the correspondences between $\mathsf{ticked}_{\eta}^{\kappa}(q)$, $\mathsf{cl}_{\eta}(q)$, and $\mathsf{cl}(q[\eta])$ precise:

Proposition 5.10 (ticked-cl correspondence). *If* $\langle \cdot \checkmark \eta / \Delta \rangle \overset{\kappa \mapsto t}{\Longmapsto}{}^{*} \langle \varepsilon \rangle$ *and* $\langle s; \varepsilon \rangle \Downarrow \langle p'; \eta_N \checkmark \eta_E / \Delta'' \rangle$ *with* $\vdash_{\Delta} \eta : \mathsf{now}$, *then the correspondence* (1) *holds for any* $\vdash_{\Delta}^{|\eta_N|} q : \textcircled{\exists} A$.

That is, after any sequence of updates $\langle \cdot \checkmark \eta / \Delta \rangle \overset{\kappa \mapsto t}{\Longmapsto}{}^{*} \langle \varepsilon \rangle$ and any further allocations in ε performed by arbitrary evaluation $\langle s; \varepsilon \rangle \Downarrow \langle p'; \eta_N \checkmark \eta_E / \Delta'' \rangle$, the correspondence (1) between $\mathsf{ticked}_{\eta_N}^{\kappa}(\cdot)$ and $\mathsf{cl}_{\eta}(\cdot)$ holds. This characterisation covers all environments $\eta_N \checkmark \eta_E / \Delta''$ that the machine may encounter during a reactive step $\langle p; \eta / \Delta \rangle \overset{\kappa \mapsto t}{\Longrightarrow} \langle p; \eta' / \Delta' \rangle$. According to Proposition 5.2 the machine preserves the typing of terms and heaps, and thus the above correspondence holds at any time of the machine's execution. Proposition 5.10 confirms that the machine faithfully implements the timing of delayed computations captured by the notion of clocks. In addition, the two definitions of clocks ($\mathsf{cl}_{\eta}(p)$ and $\mathsf{cl}(v)$) agree for well-typed values and heaps:

Proposition 5.11 (clock correspondence). *If* $\vdash_{\Delta}^{\eta} p : \textcircled{\exists} A$, *then* $(\mathsf{cl}_{\eta}(p))[\eta] = \mathsf{cl}(p[\eta])$.

6 Related Work

Modal FRP. The use of modal types for FRP was first proposed by Krishnaswami and Benton [2011a,b], and its connection to linear temporal logic was independently discovered by Jeltsch [2012] and Jeffrey [2012]. Krishnaswami [2013] was the first to exploit modal types to construct a higher-order FRP language that provably does not suffer from space leaks. This work was later extended by Bahr et al. [2019] to simplify the type system using Fitch-style tokens in the typing context [Clouston 2018]. However, unlike Rizzo, these and subsequent languages [Bahr 2022; Bahr et al. 2021] with formal guarantees about space leaks are *synchronous*, i.e. there is a single global clock and all signals update according to this global clock. Asynchronous FRP for GUIs was pioneered by Elm [Czaplicki and Chong 2013], which processes signals asynchronously but gives no static guarantees ruling out space leaks. Modal type systems were later brought to the asynchronous setting, including languages aimed specifically at GUIs [Disch et al. 2026; Graulund et al. 2021] as well as the Async RaTT calculus [Bahr and Møgelberg 2023]. To our knowledge, only Async RaTT makes formal operational guarantees including productivity and the absence of space leaks. We refer back to section 3.3 for a detailed comparison of Rizzo and Async RaTT.

Signals as Mutable Cells. The usual proof technique to establish formal guarantees about space leaks in modal FRP (first introduced by Krishnaswami [2013], see above) is based on an operational semantics that has a special heap to store delayed computations and evicts such delayed computations from the heap after each tick of the global clock. By contrast, we use a special heap for signals rather than delayed computations, and our operational semantics ensures that we can only store the most recent value of each signal. The representation of FRP signals as mutable cells in a dataflow graph is found in other FRP implementations such as FrTime [Cooper and Krishnamurthi 2006] and Flapjax [Meyerovich et al. 2009], as well as in the original work of Krishnaswami and Benton [2011a,b] on modal FRP: Although their denotational semantics defines signals as infinite streams, which may suffer from space leaks, Krishnaswami and Benton also give a DSL

that implements signals as mutable cells in a dataflow graph – similar to Rizzo's heap of stored signals, albeit in a synchronous setting. Later modal FRP languages, which establish guarantees about space leaks in the style of Krishnaswami [2013], did not pick up this representation. But it is standard in imperative reactive programming approaches and now underpins mainstream reactive user-interface frameworks such as SolidJS, Vue, Angular, Svelte, and the TC39 Signals proposal for JavaScript [TC39 2024]. Unlike FRP languages that use mutable cells merely as an efficient underlying implementation of read-only signals, these imperative frameworks expose signals as *writable* cells that can be mutated through setters – typically from event callbacks.

FRP as a Library. Operational properties of reactive programs can also be ensured by devising a library with a carefully restricted interface. For example, Yampa [Nilsson et al. 2002] uses arrows [Hughes 2000] to ensure causality of signal functions, and FRPNow! [Ploeg and Claessen 2015] uses a monadic interface to avoid some sources of space leaks. A more recent refinement of Yampa [Bärenz and Perez 2018] annotates signal functions with type-level clocks, which allows the construction of dataflow graphs that combine subsystems running at different clock speeds. These type-level clocks are *statically* determined at compile time with the aim of providing efficient resampling between subsystems, whereas Rizzo's clocks may *dynamically* change over time to allow for programs with dynamically changing dataflows.

Synchronous Languages. The set of crucial operational properties we expect from reactive programs depends on the application domain. For example, most programming languages do not enforce termination, and similarly we may sacrifice the productivity guarantee in order to simplify Rizzo by allowing general recursion. However, other application domains require *stronger* guarantees such as bounded memory usage and real-time guarantees for each computation step. For example, Krishnaswami et al. [2012] use a linear typing discipline to obtain static memory bounds for FRP programs. Moreover, synchronous dataflow languages such as Esterel [Berry and Cosserat 1985; Berry and Gonthier 1992], Lustre [Caspi et al. 1987], and Lucid Synchrone [Pouzet 2006] provide static bounds on runtime. However, these languages are all limited to a synchronous setting where all signals are updated according to a global clock. Moreover, synchronous dataflow languages obtain their strong static guarantees by enforcing strict limits on the dynamic behaviour, disallowing both time-varying values of arbitrary types (e.g. a signal of signals is not allowed) and dynamic switching (i.e. no functionality equivalent to the *switch* combinator). Both Lustre and Lucid Synchrone have a notion of a local clock, which is a stream of Booleans that indicates, at each tick of the *global* clock, whether the local clock ticks as well. In Rizzo, this notion of local clocks is subsumed by partial signals, which give rise to clocks via $\mathsf{watch} : \mathsf{Sig}\ (A + 1) \to \Circledcirc A$.

7 Conclusion and Future Work

Rizzo takes a different approach to asynchronous modal FRP. Similarly to the asynchronous modal FRP languages Async RaTT [Bahr and Møgelberg 2023] and $\lambda_{\mathsf{Widget}}$ [Graulund et al. 2021], it uses modal types to keep track of time. However, unlike these languages, Rizzo combines modal types with a forgetful semantics so that signals can be updated in place. This simplifies the type system and extends the expressiveness of the language, all while maintaining the operational guarantees of causality, productivity, and absence of space leaks.

As discussed in section 2.4, β- and η-equality do not hold universally in Async RaTT and Rizzo. This suggests future work in studying the equational theory of asynchronous FRP and devising a logic for convenient equational reasoning.

Acknowledgements

We thank Patrick Bohn Matthiesen and Tobias Skovbæk Brund for useful comments and suggestions that improved this article.

References

Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

Patrick Bahr. 2026. Lean formalisation of Rizzo. https://github.com/pa-ba/rizzo-metatheory.

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming Without Space Leaks. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 109:1–109:27. doi:10.1145/3341713

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proceedings of the ACM on Programming Languages* 5 (2021), 2:1–2:28. Issue POPL. doi:10.1145/3434283

Patrick Bahr and Rasmus Ejlers Møgelberg. 2023. Asynchronous Modal FRP. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 205:476–205:510. doi:10.1145/3607847

Gérard Berry and Laurent Cosserat. 1985. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*. 389–448. doi:10.1007/3-540-15670-4_19

Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. doi:10.1016/0167-6423(92)90005-V

Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. 145–157. doi:10.1145/3242744.3242757

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 178–188. doi:10.1145/41625.41641

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures*. 258–275.

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems*. 294–308. doi:10.1007/11693024_20

Antony Courtney and Conal Elliott. 2001. Genuinely functional user interfaces. In *Proceedings of the 2001 Haskell Workshop*.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 411–422. doi:10.1145/2499370.2462161

Jean-Claude Disch, Asger Heegaard, and Patrick Bahr. 2026. Functional Reactive GUI Programming with Modal Types. In *Trends in Functional Programming*. 93–114. doi:10.1007/978-3-031-99751-8_5

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 263–273. doi:10.1145/258948.258973

Christian Uldal Graulund, Dmitrij Szamozvancev, and Neel Krishnaswami. 2021. Adjoint Reactive GUI Programming. In *Foundations of Software Science and Computation Structures*. 289–309. doi:10.1007/978-3-030-71995-1_15

John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

Alan Jeffrey. 2012. 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*. 49–60. doi:10.1145/2103776.2103783

Wolfgang Jeltsch. 2012. Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming. *Electronic Notes in Theoretical Computer Science* 286 (2012), 229–242. doi:10.1016/j.entcs.2012.08.015

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. 221–232. doi:10.1145/2500365.2500588

Neelakantan R. Krishnaswami and Nick Benton. 2011a. A semantic model for graphical user interfaces. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. 45–57. doi:10.1145/2034773.2034782

Neelakantan R. Krishnaswami and Nick Benton. 2011b. Ultrametric Semantics of Reactive Programs. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*. 257–266. doi:10.1109/LICS.2011.38

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order Functional Reactive Programming in Bounded Space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 45–58. doi:10.1145/2103656.2103665

Anton Lorenzen and Daan Leijen. 2022. Reference counting with frame limited reuse. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 103:357–103:380. doi:10.1145/3547634

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. doi:10.1017/S0956796807006326

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 1–20. doi:10.1145/1640089.1640091

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*. 625–635. doi:10.1007/978-3-030-79876-5_37

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*. 255–266. doi:10.1109/LICS.2000.855774

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. 51–64. doi:10.1145/581690.581695

Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 302–314. doi:10.1145/2784731.2784752

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. 46–57. doi:10.1109/SFCS.1977.32

Marc Pouzet. 2006. *Lucid Synchrone, version 3*. Technical Report hal-03090137. 25 pages. https://hal.science/hal-03090137v1

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 96–111. doi:10.1145/3453483.3454032

TC39. 2024. Signals: A Proposal to Add Signals to ECMAScript. https://github.com/tc39/proposal-signals. ECMAScript proposal, Stage 1. Accessed 2026-07-03.

Sebastian Ullrich and Leonardo de Moura. 2021. Counting immutable beans: reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 1–12. doi:10.1145/3412932.3412935

Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. 242–252. doi:10.1145/349299.349331

A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093

A Example Runs of the Machine

A.1 Sample

Our example program t assumes a channel context $\Delta = \{\kappa_1 : \mathsf{Chan}\ \mathsf{Nat}, \kappa_2 : \mathsf{Chan}\ \mathit{Char}\}$:

$$t = \mathbf{let}\ \mathit{xs} = 0 :: \mathit{mkSig}\ (\mathsf{wait}\ \kappa_1)\ \mathbf{in}\ \mathbf{let}\ \mathit{ys} = \texttt{'a'} :: \mathit{mkSig}\ (\mathsf{wait}\ \kappa_2)\ \mathbf{in}\ \mathit{sample}\ \mathit{xs}\ \mathit{ys}$$

This program first constructs two signals xs and ys from the two channels κ_1 and κ_2 and then samples from ys using xs. Before we run t, we translate the definitions of *mkSig* and *sample* (as given in section 2.2) from the surface syntax into the core calculus, similarly to how we translate *map* at the end of section 2.3:

$$\begin{aligned}\mathit{sample} &= \lambda \mathit{xs}.\lambda \mathit{ys}.\mathit{map}\ (\lambda x.(x, \mathsf{head}\ \mathit{ys}))\ \mathit{xs}\\ \mathit{mkSig} &= \mathsf{fix}\ r.\lambda d.\mathsf{delay}\ (\lambda r'.\lambda x.x :: r'\ d) \circledast r \oslash d\end{aligned}$$

Assuming two signals l_1 and l_2, the term $\mathit{sample}\ l_1\ l_2$ evaluates to a signal with head (p_1, p_2), where each p_i is the head of l_i, and the following tail sp:

$$\mathit{sp} = \mathsf{delay}\ ((\lambda r'.r'\ (\lambda x.(x, \mathsf{head}\ l_2)))\ \mathit{map}) \oslash \mathsf{tail}\ l_1$$

By definition, $\mathsf{cl}_\eta\ (\mathit{sp}) = \mathsf{cl}_\eta\ (\mathsf{tail}\ l_1)$, and thus we know that the signal produced by *sample* updates whenever the signal l_1 updates.

In turn, assuming a value $d : \Box\, A$, the term $\mathit{mkSig}\ d$ evaluates to the following value $\mathit{sig}[d]$:

$$\mathit{sig}[d] = \mathsf{delay}\ ((\lambda r'.\lambda x.x :: r'\ d)\ \mathit{mkSig}) \oslash d$$

The notation $\mathit{sig}[d]$ is meant to indicate that d is a placeholder. For example, $\mathit{mkSig}\ (\mathsf{wait}\ \kappa_1)$ evaluates to $\mathit{sig}[\mathsf{wait}\ \kappa_1] = \mathsf{delay}\ ((\lambda r'.\lambda x.x :: r'\ (\mathsf{wait}\ \kappa_1))\ \mathit{mkSig}) \oslash \mathsf{wait}\ \kappa_1$.

Note that, according to the evaluation semantics for $\circledast$, the function application that appears as an argument to delay in both sp and $\mathit{sig}[d]$ has not been reduced any further. However, we can β-reduce both and obtain semantically equivalent terms that are easier to read:

$$\begin{aligned}\mathit{sp} &\equiv \mathsf{delay}\ (\mathit{map}\ (\lambda x.(x, \mathsf{head}\ l_2))) \oslash \mathsf{tail}\ l_1\\ \mathit{sig}[d] &\equiv \mathsf{delay}\ (\lambda x.x :: \mathit{mkSig}\ d) \oslash d\end{aligned}$$

With these observations, we can now see how t reacts to alternating events from the two channels in Δ. To avoid clutter, we elide heap locations that are not referenced anywhere and also leave out the type annotations of all heap locations:

$$\begin{aligned}\langle t;\Delta\rangle &\overset{\text{init}}{\Longrightarrow} \langle l_3; l_1 \mapsto 0\,\langle\bot\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_1], l_2 \mapsto \texttt{'a'}\,\langle\bot\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_2], l_3 \mapsto (0, \texttt{'a'})\,\langle\bot\rangle\ \mathit{sp}/\Delta\rangle\\ &\overset{\kappa_1\mapsto 1}{\Longrightarrow} \langle l_3; l_1 \mapsto 1\,\langle\top\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_1], l_2 \mapsto \texttt{'a'}\,\langle\bot\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_2], l_3 \mapsto (1, \texttt{'a'})\,\langle\top\rangle\ \mathit{sp}/\Delta\rangle\\ &\overset{\kappa_2\mapsto \texttt{'b'}}{\Longrightarrow} \langle l_3; l_1 \mapsto 1\,\langle\bot\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_1], l_2 \mapsto \texttt{'b'}\,\langle\top\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_2], l_3 \mapsto (1, \texttt{'a'})\,\langle\bot\rangle\ \mathit{sp}/\Delta\rangle\\ &\overset{\kappa_1\mapsto 2}{\Longrightarrow} \langle l_3; l_1 \mapsto 2\,\langle\top\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_1], l_2 \mapsto \texttt{'b'}\,\langle\bot\rangle\ \mathit{sig}[\mathsf{wait}\ \kappa_2], l_3 \mapsto (2, \texttt{'b'})\,\langle\top\rangle\ \mathit{sp}/\Delta\rangle\end{aligned}$$

The initialisation step allocates three signals on the heap. Each of the first two, at l_1 and l_2, updates whenever the corresponding channel κ_i receives an input since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_i) = \mathsf{cl}_\eta\ (\mathit{sig}[\mathsf{wait}\ \kappa_i]) = \mathsf{cl}_\eta\ (\mathsf{wait}\ \kappa_i) = \{\kappa_i\}$. The signal at l_3, however, only updates whenever κ_1 receives an input since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_3) = \mathsf{cl}_\eta\ (\mathit{sp}) = \mathsf{cl}_\eta\ (\mathsf{tail}\ l_1) = \{\kappa_1\}$. That is, we have the following reactive evaluation for the sequence of events $\tau = \kappa_1 \mapsto 1, \kappa_2 \mapsto \texttt{'b'}, \kappa_1 \mapsto 2$:

$$\langle t;\Delta\rangle \rightsquigarrow_\tau \langle (2, \texttt{'b'}) :: \mathsf{delay}\ (\mathit{map}\ (\lambda x.(x, \mathsf{head}\ (\texttt{'b'} :: \mathit{sig}[\mathsf{wait}\ \kappa_2])))) \oslash \mathit{sig}[\mathsf{wait}\ \kappa_1]\,;\Delta\rangle$$

A.2 Filter

We assume a channel context $\Delta = \{\kappa_1 : \mathsf{Chan}\ \mathsf{Nat}\}$ and a function $isEven : \mathsf{Nat} \to Bool$ that checks whether a number is even:

$$t = \mathbf{let}\ xs = mkSig\ (\mathsf{wait}\ \kappa_1)\ \mathbf{in}\ 0 :: filter\ isEven\ xs$$

That is, t constructs a delayed signal from κ_1 and then filters that signal using the *isEven* predicate. To construct its result, *filter* takes *isEven* and constructs a new function p:

$$p = \lambda x.\mathbf{if}\ isEven\ x\ \mathbf{then}\ \mathtt{just}\ x\ \mathbf{else}\ \mathtt{nothing}$$

Assuming the channel κ_1 receives increasing numbers, t produces the following behaviour:

$$\begin{aligned}
\langle t;\Delta\rangle &\overset{\mathsf{init}}{\Longrightarrow} \langle l_2; l_1 \mapsto \mathtt{nothing}\,\langle\bot\rangle\ \mathsf{delay}\ (map\ p) \ominus sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 0\,\langle\bot\rangle\ sig[\mathsf{watch}\ l_1]/\Delta\rangle \\
&\overset{\kappa_1\mapsto 1}{\Longrightarrow} \langle l_2; l_3 \mapsto 1\,\langle\bot\rangle\ sig[\mathsf{wait}\ \kappa_1], l_1 \mapsto \mathtt{nothing}\,\langle\top\rangle\ \mathsf{delay}\ ((\lambda r'.r'\ p)\ map) \ominus \mathsf{tail}\ l_3, \\
&\qquad l_2 \mapsto 0\,\langle\bot\rangle\ sig[\mathsf{watch}\ l_1]/\Delta\rangle \\
&\overset{\kappa_1\mapsto 2}{\Longrightarrow} \langle l_2; l_3 \mapsto 2\,\langle\top\rangle\ sig[\mathsf{wait}\ \kappa_1], l_1 \mapsto \mathtt{just}\ 2\,\langle\top\rangle\ \mathsf{delay}\ ((\lambda r'.r'\ p)\ map) \ominus \mathsf{tail}\ l_3, \\
&\qquad l_2 \mapsto 2\,\langle\top\rangle\ sig[\mathsf{watch}\ l_1]/\Delta\rangle
\end{aligned}$$

The initialisation step allocates two signals. The signal of type Sig (*Maybe* Nat) constructed by *filter isEven xs* using $\mathtt{nothing} :: (map\ p \triangleright xs)$ is stored at l_1. The signal of type Sig Nat stored at l_2 is the result of evaluating $0 :: mkSig\ (\mathsf{watch}\ l_1)$. When the first input on κ_1 arrives, signal l_1 needs updating since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_1) = \mathsf{cl}_\eta\ (sig[\mathsf{wait}\ \kappa_1]) = \mathsf{cl}_\eta\ (\mathsf{wait}\ \kappa_1) = \{\kappa_1\}$. This requires the delayed computation $sig[\mathsf{wait}\ \kappa_1]$ to be performed by the advance semantics, which in turn results in the allocation of the signal at l_3. However, the signal at l_2 is not updated since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_2) = \mathsf{cl}_\eta\ (sig[\mathsf{watch}\ l_1]) = \mathsf{cl}_\eta\ (\mathsf{watch}\ l_1) = \{l_1\}$, and while the signal at l_1 was updated (indicated by $\top$), its current value is not of the form $\mathtt{just}\ q$. Only after receiving the next input on channel κ_1, the signal at l_1 is updated to have current value $\mathtt{just}\ 2$ and thus also l_2 is updated. That is, the events $\tau = \kappa_1 \mapsto 1, \kappa_1 \mapsto 2$ cause the following reactive evaluation:

$$\langle t;\Delta\rangle \rightsquigarrow_\tau \langle 2 :: sig[\mathsf{watch}\ (\mathtt{just}\ 2 :: \mathsf{delay}\ ((\lambda r'.r'\ p)\ map) \ominus sig[\mathsf{wait}\ \kappa_1])]\,;\Delta\rangle$$

A.3 Switch

We assume a channel context $\Delta = \{\kappa_1 : \mathsf{Chan}\ \mathsf{Nat}, \kappa_2 : \mathsf{Chan}\ \mathsf{Nat}\}$ to implement the following example involving *switch*:

$$t = \mathbf{let}\ xs = 0 :: mkSig\ (\mathsf{wait}\ \kappa_1)\ \mathbf{in}\ \mathbf{let}\ ys = mkSig\ (\mathsf{wait}\ \kappa_2)\ \mathbf{in}\ switch\ xs\ ys$$

That is, the signal produced by t first behaves like the signal produced by κ_1, but then behaves as the signal produced by κ_2 as soon as κ_2 receives its first input.

We again translate the definition of *switch* into the core calculus of Rizzo, similarly to how we translate *map* and *sample*:

$$\begin{aligned}
switch &= \mathsf{fix}\ r.\lambda s.\lambda d.\mathbf{let}\ x = \mathsf{head}\ s\ \mathbf{in}\ \mathbf{let}\ xs = \mathsf{tail}\ s\ \mathbf{in}\ x :: (\mathsf{delay}\ cont \circledast r \ominus \mathsf{select}\ xs\ d) \\
cont &= \lambda r'.\lambda y.\mathbf{case}\ y\ \mathbf{of}\ \{\mathtt{left}\ z.r'\ z\ d; \mathtt{right}\ d'.d'; \mathtt{both}\ u\ d'.d'\}
\end{aligned}$$

Assuming a signal l_1 and a delayed signal $d : \ominus$ (Sig Nat), the term $switch\ l_1\ d$ evaluates to a signal whose head is the head of l_1 and whose tail sw is

$$sw = \mathsf{delay}\ (cont\ switch) \ominus \mathsf{select}\ (\mathsf{tail}\ l_1)\ d,$$

which we may β-reduce to the more readable

$$sw \equiv \mathsf{delay}\ cont' \circledcirc \mathsf{select}\ (\mathsf{tail}\ l_1)\ d$$
$$cont' = \lambda y.\mathbf{case}\ y\ \mathbf{of}\ \{\mathtt{left}\ z.switch\ z\ d; \mathtt{right}\ d'.d'; \mathtt{both}\ u\ d'.d'\}$$

By definition, $\mathsf{cl}_\eta\ (sw) = \mathsf{cl}_\eta\ (\mathsf{select}\ (\mathsf{tail}\ l_1)\ d) = \mathsf{cl}_\eta\ (\mathsf{tail}\ l_1) \cup \mathsf{cl}_\eta\ (d)$. If d is the result of evaluating $mkSig\ (\mathsf{wait}\ \kappa_2)$, namely $d = sig[\mathsf{wait}\ \kappa_2]$, then $\mathsf{cl}_\eta\ (d) = \{\kappa_2\}$, and $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_1) = \{\kappa_1\}$; hence $\mathsf{cl}_\eta\ (sw) = \{\kappa_1, \kappa_2\}$, i.e. the switched signal updates whenever *either* l_1 (the current signal) or d (the switch trigger) ticks.

As in previous examples, we elide heap locations (except l_1) that are not referenced anywhere as well as the type annotations of heap locations. The program t then reacts to the events $\kappa_1 \mapsto 1, \kappa_1 \mapsto 2, \kappa_2 \mapsto 5, \kappa_1 \mapsto 3$ as follows:

$$\begin{aligned}
\langle t; \Delta\rangle &\overset{\mathsf{init}}{\Longrightarrow} \langle l_2; l_1 \mapsto 0\,\langle\bot\rangle\ sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 0\,\langle\bot\rangle\ sw/\Delta\rangle \\
&\overset{\kappa_1\mapsto 1}{\Longrightarrow} \langle l_2; l_1 \mapsto 1\,\langle\top\rangle\ sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 1\,\langle\top\rangle\ sw/\Delta\rangle \\
&\overset{\kappa_1\mapsto 2}{\Longrightarrow} \langle l_2; l_1 \mapsto 2\,\langle\top\rangle\ sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 2\,\langle\top\rangle\ sw/\Delta\rangle \\
&\overset{\kappa_2\mapsto 5}{\Longrightarrow} \langle l_2; l_1 \mapsto 2\,\langle\bot\rangle\ sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 5\,\langle\top\rangle\ sig[\mathsf{wait}\ \kappa_2]/\Delta\rangle \\
&\overset{\kappa_1\mapsto 3}{\Longrightarrow} \langle l_2; l_1 \mapsto 3\,\langle\top\rangle\ sig[\mathsf{wait}\ \kappa_1], l_2 \mapsto 5\,\langle\bot\rangle\ sig[\mathsf{wait}\ \kappa_2]/\Delta\rangle
\end{aligned}$$

The initialisation step allocates two signals: the initial signal xs at l_1 and the *switch* result at l_2 (the result of evaluating t). The signal at l_1 updates whenever κ_1 receives an input, since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_1) = \{\kappa_1\}$, whereas the signal at l_2 updates whenever κ_1 *or* κ_2 receives an input, since $\mathsf{cl}_\eta\ (\mathsf{tail}\ l_2) = \mathsf{cl}_\eta\ (sw) = \{\kappa_1, \kappa_2\}$.

For the first two events $\kappa_1 \mapsto 1$ and $\kappa_1 \mapsto 2$, both signals tick: the select in sw fires on its left branch, so *switch* recurses and l_2 keeps the tail sw while its head follows l_1. The event $\kappa_2 \mapsto 5$ then *fires the switch*: l_2 ticks (as $\kappa_2 \in \mathsf{cl}_\eta\ (sw)$) but l_1 does not (as $\kappa_2 \notin \{\kappa_1\}$), and so the select fires on its right branch, where $cont'$ returns the delivered signal d'. From this point on l_2 *is* the signal ys: its head becomes 5 and its tail becomes $sig[\mathsf{wait}\ \kappa_2]$, whose clock is $\{\kappa_2\}$. Consequently, the final event $\kappa_1 \mapsto 3$ updates only l_1; the switched signal at l_2 no longer reacts to κ_1 and retains the value 5.

That is, the events $\tau = \kappa_1 \mapsto 1, \kappa_1 \mapsto 2, \kappa_2 \mapsto 5, \kappa_1 \mapsto 3$ cause the following reactive evaluation:

$$\langle t; \Delta\rangle \rightsquigarrow_\tau \langle 5 :: sig[\mathsf{wait}\ \kappa_2]\,; \Delta\rangle\,.$$

B *switchAt* in Async RaTT

We implement the *switchAt* combinator introduced in section 3.3.1 using Async RaTT [Bahr and Møgelberg 2023]. To this end, we implement two helper functions. The first one *del* takes a signal $s : \mathsf{Sig}\ C$ and a delayed unit $d : \Leftarrow 1$ and tries to delay s until after d ticks. If this fails, i.e. if the tail of s arrives before d, then the returned *Maybe* value contains `just` d (but moved into the future where the tail of s ticked). Otherwise, the *Maybe* value is `nothing`:

$$\begin{aligned}
&del : stable\ C \Rightarrow \mathsf{Sig}\ C \to \Leftarrow 1 \to \Leftarrow (Maybe\ (\Leftarrow 1) \times \mathsf{Sig}\ C) \\
&del\ (x :: xs)\ d = \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ d\ \mathbf{of} \\
&\qquad\qquad \mathtt{left}\ \ xs'\ d'.(\mathtt{just}\ d', xs') \\
&\qquad\qquad \mathtt{right}\ xs'\ _.(\mathtt{nothing}, x :: xs') \\
&\qquad\qquad \mathtt{both}\ \ xs'\ _.(\mathtt{nothing}, xs'))
\end{aligned}$$

We follow the convention of Bahr and Møgelberg [2023] and write *stable* C to indicate that the above function requires that the type C is stable. Note also that, unlike Rizzo's select, the select of

Async RaTT returns in its `left` (resp. `right`) branch not only the value produced by the first (resp. second) argument but also the still-delayed computation of the other argument. This is why, for example, the `left` branch of *del* binds both the new signal xs' and the still-pending delay d' of d.

The function *switchN* implements the main logic of *switchAt*. It takes a signal $xs : \mathsf{Sig}\ C$ and a delayed value $e : \textcircled{\exists}\ (\mathit{Maybe}\ (\textcircled{\exists}\ 1) \times \mathsf{Sig}\ C)$ (which is the result of *del ys d*) and produces a signal that first behaves like *xs* but switches to *ys* as soon as *d* arrives. To this end, it repeatedly calls *del* if the delayed value *e* produces a `just` value, which indicates that *d* has not arrived yet:

$$
\begin{aligned}
&\mathit{switchN} : \mathit{stable}\ C \Rightarrow \mathsf{Sig}\ C \to \textcircled{\exists}\ (\mathit{Maybe}\ (\textcircled{\exists}\ 1) \times \mathsf{Sig}\ C) \to \mathsf{Sig}\ C \\
&\mathit{switchN}\ (x :: xs)\ e = x :: \mathsf{delay}\ (\textbf{case}\ \mathsf{select}\ xs\ e\ \textbf{of} \\
&\qquad\qquad \texttt{left}\ \ xs'\ e' \qquad\qquad\quad .\mathit{switchN}\ xs'\ e' \\
&\qquad\qquad \texttt{right}\ xs'\ (\texttt{just}\ d', ys')\ .\mathit{switchN}\ (x :: xs')\ (\mathit{del}\ ys'\ d') \\
&\qquad\qquad \texttt{right}\ xs'\ (\texttt{nothing}, ys').ys' \\
&\qquad\qquad \texttt{both}\ \ xs'\ (\texttt{just}\ d', ys')\ .\mathit{switchN}\ xs'\ (\mathit{del}\ ys'\ d') \\
&\qquad\qquad \texttt{both}\ \ xs'\ (\texttt{nothing}, ys').ys') \\
&\mathit{switchAt} : \mathit{stable}\ C \Rightarrow \mathsf{Sig}\ C \to \mathsf{Sig}\ C \to \textcircled{\exists}\ 1 \to \mathsf{Sig}\ C \\
&\mathit{switchAt}\ xs\ ys\ d = \mathit{switchN}\ xs\ (\mathit{del}\ ys\ d)
\end{aligned}
$$

Note that this implementation has a less general type than the corresponding Rizzo implementation in section 3.3.1 as it requires that C be stable.