Asynchronous Modal FRP

PATRICK BAHR, IT University of Copenhagen, Denmark
RASMUS EJLERS MØGELBERG, IT University of Copenhagen, Denmark

Over the past decade, a number of languages for functional reactive programming (FRP) have been suggested, which use modal types to ensure properties like causality, productivity and lack of space leaks. So far, almost all of these languages have included a modal operator for delay on a global clock. For some applications, however, a global clock is unnatural and leads to leaky abstractions as well as inefficient implementations. While modal languages without a global clock have been proposed, no operational properties have been proved about them, yet.

This paper proposes Async RaTT, a new modal language for asynchronous FRP, equipped with an operational semantics mapping complete programs to machines that take asynchronous input signals and produce output signals. The main novelty of Async RaTT is a new modality for asynchronous delay, allowing each output channel to be associated at runtime with the set of input channels it depends on, thus causing the machine to only compute new output when necessary. We prove a series of operational properties including causality, productivity and lack of space leaks. We also show that, although the set of input channels associated with an output channel can change during execution, upper bounds on these can be determined statically by the type system.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Data flow languages**; *Recursion*; • **Theory of computation** → *Operational semantics*.

Additional Key Words and Phrases: Functional Reactive Programming, Modal Types, Linear Temporal Logic, Synchronous Data Flow Languages, Type Systems

ACM Reference Format:
Patrick Bahr and Rasmus Ejlers Møgelberg. 2023. Asynchronous Modal FRP. *Proc. ACM Program. Lang.* 7, ICFP, Article 205 (August 2023), 35 pages. https://doi.org/10.1145/3607847

1 INTRODUCTION

Reactive programs are programs that engage in a dialogue with their environment, receiving input and producing output, often without ever terminating. Examples include much of the most safety critical software in use today, such as control software and servers, as well as GUIs. Most reactive software is written in imperative languages using a combination of complex features such as callbacks and shared memory, and for this reason it is error-prone and hard to reason about.

The idea of functional reactive programming (FRP) [Elliott and Hudak 1997] is to provide the programmer with the right abstractions to write reactive programs in functional style, allowing for short modular programs, as well as modular reasoning about these. For such abstractions to be useful it is important that they are designed to allow for efficient low-level implementations to be automatically generated from programs.

The main abstraction of FRP is that of signals, which are time-dependent values. In the case of discrete time given by a global clock, a signal can be thought of as a stream of data. A reactive

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark, paba@itu.dk; Rasmus Ejlers Møgelberg, IT University of Copenhagen, Denmark, mogel@itu.dk.

2475-1421/2023/8-ART205
https://doi.org/10.1145/3607847

program is essentially just a function taking input signals and producing output signals. For this to be implementable, however, it needs to be causal: The current output must only depend on current and past input. Moreover, the low-level implementations generated from high-level programs should also be free of (implicit) space- and time-leaks. This means that reactive programs should not store data indefinitely causing the program to eventually run out of space, nor should they repeat computations in such a way that the execution of each step becomes increasingly slower.

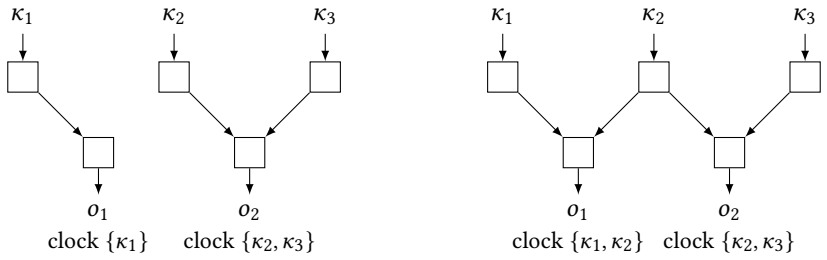
These requirements have led to the development of modal FRP [Bahr 2022; Bahr et al. 2019; Jeffrey 2012, 2014; Jeltsch 2012; Krishnaswami 2013; Krishnaswami and Benton 2011; Krishnaswami et al. 2012], a family of languages using modal types to ensure that all programs can be implemented efficiently. The most important modal type constructor is $\bigcirc$, used to classify data available in the next time step on some global discrete clock. For example, the type of signals should satisfy the type isomorphism $\mathrm{Sig}\, A \cong A \times \bigcirc(\mathrm{Sig}\, A)$ stating that the current value of the signal is available now, but its future values are only available after the next time step. Using this encoding of signals, one can ensure that all reactive programs are causal. Many modal FRP languages also include a variant of the Nakano [2000] guarded fixed point operator of type $(\bigcirc A \to A) \to A$. The type ensures that recursive calls are only performed in future steps, thus ensuring termination of each step of computation, a property called *productivity*. Often these languages also include a $\Box$ modality used to classify data that is *stable*, in the sense that it can be kept until the next time step without causing space leaks. Other modal constructors, such as $\Diamond$ (eventually) can be encoded, suggesting a Curry-Howard correspondence between linear temporal logic [Pnueli 1977] and modal FRP [Bahr et al. 2021; Cave et al. 2014; Jeffrey 2012; Jeltsch 2012].

However, for many applications, the notion of a global clock associated with the $\bigcirc$ modal operator may not be natural and can also lead to inefficient implementations. Consider, for example, a GUI which takes an input signal of user keystrokes, as well as other signals that are updated more frequently, like the mouse pointer coordinates. The global clock would have to tick at least as fast as the updates to the fastest signal, and updates on the keystroke signal will only happen on very few ticks on the global clock. Perhaps the most natural way to model the keystroke signal is therefore using a signal of type $\mathsf{Maybe}(\mathsf{Char})$. In the modal FRP languages of Bahr et al. [2019]; Krishnaswami [2013], the processor for this signal will have to wake up for each tick on the global clock, check for input, and often also transport some local state to the next time step by calling itself recursively. Perhaps more problematic, however, is that an important abstraction barrier is broken when a processor for an input signal is given access to the global clock. Instead, we would like to write the GUI as a collection of processors for asynchronous input signals that are only activated upon updates to the signals on which they depend.

1.1 Async RaTT

This paper presents Async RaTT, a modal FRP language in the RaTT family [Bahr 2022; Bahr et al. 2019, 2021], designed for processing asynchronous input. A reactive program in Async RaTT reads signals from a set of *input channels* and in response sends signals to a set of *output channels*. In a GUI application, typical input channels would include the mouse position and keystroke events, while output channels could for example include the content of a text field or the colour of a text field.

For each output channel o, the reactive program keeps track of the set θ of input channels on which o depends (cf. Figure 1a). We refer to such a set θ of input channels as a *clock*. When the signal on an input channel κ is updated, only those output channels whose clock θ contains κ will be updated. For example, the keystroke input channel might be in the clock for the text field content but not the text field colour. Since the program can dynamically change its internal dataflow graph, the clock associated with an output channel may change during execution (cf. Figure 1b) and so is



(a) Dataflow graph with its computed clocks. (b) Clocks are updated as dependencies change.

Fig. 1. Dynamically changing dataflow graph of an Async RaTT program with input channels $\kappa_1, \kappa_2, \kappa_3$ and output channels o_1, o_2.

not known at compile time. For example, the text field might fall out of focus and thus not react to keystrokes any longer. We refer to the arrival of new data on an input channel in the clock θ as a *tick* on clock θ.

Async RaTT has a modal operator $\Box$ used to classify stable data, as well as two new modalities: $\textcircled{\exists}$ for asynchronous delays and $\textcircled{\forall}$ for a delay on the global clock. A value of type $\textcircled{\exists}A$ is a pair consisting of a clock θ and a computation that can be executed to return data of type A on the first tick on θ. The type $\textcircled{\exists}A$ can therefore be thought of as an existential type. The clocks of output channels, as illustrated in Figure 1, are stored in the first component of this existential type. Our notion of signal is encoded in types as a recursive type $\mathrm{Sig}\, A \cong A \times \textcircled{\exists}\mathrm{Sig}\, A$. That means, the clock associated with the tail of a signal may change from one step to the next, allowing for dynamic updates of clocks associated with output channels as in Figure 1.

Unlike the synchronous $\bigcirc$, the asynchronous $\textcircled{\exists}$ does not have an applicative action of type $\textcircled{\exists}(A \to B) \to \textcircled{\exists}A \to \textcircled{\exists}B$ because the delayed function and the delayed input may not arrive at the same time, and to avoid space leaks, Async RaTT does not allow the first input to be stored until the second input arrives. Instead, Async RaTT synchronises delayed data using an operator

$$\mathsf{sync} : \textcircled{\exists}A_1 \to \textcircled{\exists}A_2 \to \textcircled{\exists}((A_1 \times \textcircled{\exists}A_2) + (\textcircled{\exists}A_1 \times A_2) + (A_1 \times A_2))$$

Given two delayed computations associated with clocks θ_1 and θ_2, respectively, sync returns the delayed computation associated with the union clock $\theta_1 \sqcup \theta_2$. This delayed computation waits for an input on any input channel $\kappa \in \theta_1 \sqcup \theta_2$, and then evaluates the computations that can be evaluated depending on whether $\kappa \in \theta_1$, $\kappa \in \theta_2$, or both. For example, if the input arrives on channel $\kappa \in \theta_1 \setminus \theta_2$, only the first delayed computation is evaluated. The sync operator can be used to implement operators like

$$\mathsf{switch} : \mathrm{Sig}\, A \to \textcircled{\exists}(\mathrm{Sig}\, A) \to \mathrm{Sig}\, A$$

which dynamically update the dataflow graph of a program.

Note that sync can be read as a linear time axiom: Given two clocks, either one ticks before the other, or they tick simultaneously. Async RaTT programs are therefore dependent on the run-time environment to schedule the order in which inputs are processed. What we mean by asynchronicity is that output channels are updated asynchronously. This is reflected in the type system by $\textcircled{\exists}$ not being an applicative functor as explained above.

The modal type $\textcircled{\forall}A$ classifies computations that can be run at any time in the future, but not now. It is used in the guarded fixed point operator, which in Async RaTT has type

$$\Box(\textcircled{\forall}A \to A) \to A$$

The input to the fixed point operator must be a stable function (as classified by $\Box$), because it will be used in unfoldings at any time in the future. The use of $\textcircled{\triangledown}$ restricts fixed points to only unfold in the future, ensuring termination of each step of computation.

1.2 Operational Semantics and Results

We present an operational semantics mapping each complete Async RaTT program to a machine that transforms a sequence of inputs received on its input channels to a sequence of outputs on its output channels. The transformation is done in steps, processing one input at a time, producing new outputs on the affected output channels.

The operational semantics consists of two parts. The first is the evaluation semantics describing the evaluation of a term in each step of the evaluation. This takes a term and a store and returns a value and an updated store in the context of current values on input signals. The store contains delayed computations, and the evaluation semantics may run previously stored delayed computations as well as store new ones to be evaluated at a later step. The reactive semantics on the other hand, describes the machine which, at each step, locates the output signals to be updated and executes the corresponding delayed computations to produce output.

The transformation of input to output described by the operational semantics is causal by construction. We show that it is also deterministic and productive (in the sense that each step terminates and never gets stuck). We also show that the execution of an Async RaTT program is free of (implicit) space leaks. This is achieved following a technique originally due to Krishnaswami [2013]: At the end of each step of execution, the machine deletes all delayed computations that in principle could have been run in the current step – regardless of whether they actually were run. All inputs are also deleted, either at the end of the step or when the next input from the same signal arrives, depending on the kind of the specific input signal. Our results show that this aggressive garbage collection strategy is safe. Of course, the programmer can still write programs that accumulate space, but such leaks will be explicit in the source program, not implicitly introduced by the implementation of the language. (See Krishnaswami [2013] for a further discussion of implicit vs explicit space leaks.)

Finally, we show that an upper bound on the dynamic clocks associated with an output signal can be computed statically. More precisely, given an Async RaTT program consisting of a number of output signals in a given context Δ of input channels, if one of the output signals can be typed in a smaller context $\Delta' \subseteq \Delta$, then that signal will never need to update on input arriving on channels in $\Delta \setminus \Delta'$. Note that this result holds despite the existence of operators like switch, which dynamically change the dataflow graph of a program.

1.3 Overview

The paper is organised as follows: Async RaTT is presented along with its typing rules in section 2, and section 3 illustrates the expressivity of Async RaTT by developing a small library of signal combinators, along with examples that use the library for GUI programming and computing integrals and derivatives of signals. The operational semantics is defined in section 4, which also illustrates it with an example, and presents the main results. Section 5 sketches the proofs of the main results, and in particular defines the Kripke logical relation used for the proofs. Finally, section 6 and section 7 discuss related work, conclusions and future work. In addition, Appendix A gives a detailed account of the proof of the fundamental property of the Kripke logical relation.

2 ASYNC RATT

This section gives an overview of Async RaTT, referring to Figures 2 and 3 for the full specification of its syntax and typing rules.

$$
\begin{array}{lllll}
\text{Locations} & l & \in & \mathsf{Loc} \\
\text{Input Channels} & \kappa & \in & \mathsf{Chan} \\
\text{Clock Expr.} & \theta & ::= & \mathsf{cl}\,(v) \mid \theta \sqcup \theta' \\
\text{Types} & A, B & ::= & \alpha \mid 1 \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \textcircled{\exists} A \mid \textcircled{\forall} A \mid \mathsf{Fix}\,\alpha.A \mid \Box A \\
\text{Stable Types} & S, S' & ::= & 1 \mid \mathsf{Nat} \mid S \times S' \mid S + S' \mid \textcircled{\forall} A \mid \Box A \\
\text{Value Types} & T, T' & ::= & 1 \mid \mathsf{Nat} \mid T \times T' \mid T + T' \\
\text{Values} & v, w & ::= & x \mid \langle\rangle \mid 0 \mid \mathsf{suc}\,v \mid \lambda x.t \mid (v, w) \mid \mathsf{in}_i\,v \mid l \mid \mathsf{wait}_\kappa \mid \mathsf{box}\,t \mid \mathsf{dfix}\,x.t \mid \mathsf{into}\,v \\
\text{Terms} & s, t & ::= & v \mid \mathsf{suc}\,t \mid \mathsf{rec}_{\mathsf{Nat}}(s, x\,y.t, u) \mid (s, t) \mid \mathsf{in}_i\,t \mid \pi_i\,t \mid t_1 t_2 \mid \mathsf{let}\,x = s\,\mathsf{in}\,t \\
& & & \mid \mathsf{case}\,t\,\mathsf{of}\,\mathsf{in}_1\,x.t_1; \mathsf{in}_2\,x.t_2 \mid \mathsf{delay}_\theta\,t \mid \mathsf{adv}\,t \mid \mathsf{select}\,v_1\,v_2 \mid \mathsf{unbox}\,t \\
& & & \mid \mathsf{fix}\,x.t \mid \mathsf{never} \mid \mathsf{into}\,t \mid \mathsf{out}\,t \mid \mathsf{read}_\kappa
\end{array}
$$

Fig. 2. Syntax.

An Async RaTT program has access to a set of input channels, each of which receive updates asynchronously from each other. To account for this, typing judgements are relative to an input channel context Δ or *input context* for short. An example of such a context is

$$\mathsf{keyPressed} :_{\mathsf{p}} \mathsf{Nat}, \mathsf{mouseCoord} :_{\mathsf{bp}} \mathsf{Nat} \times \mathsf{Nat}, \mathsf{time} :_{\mathsf{b}} \mathsf{Float}$$

There are three classes of input channels, each corresponding to one of the subscripts p, b, and bp as in the example above. *Push-only* input channels, indicated by p, are input channels whose updates are pushed through the program, possibly causing output channels to be updated. In the example context above, the programmer will want to react to user keypresses immediately, and so updates to this should be pushed. On the other hand, we may wish to have access to a time input channel, which we can read from at any time, but we may not want the program to wake up whenever the time changes. Time is therefore treated as a *buffered-only* input channel, indicated by b, whose most recent value is buffered, but whose changes will not trigger the program to update any output channel. Finally, input channels may be both buffered and pushed, indicated by bp, which means that updates are pushed, but we also keep the value around in a buffer, so that the latest value can always be read by the program. This is unlike the push-only input channels whose values are deleted for space efficiency reasons, once an update push has been treated. For example, we might want to be informed when the mouse coordinates are updated, but also keep these around so that we can read the mouse coordinates when a key is pressed, even if the mouse has not moved. We refer to input channels that are either push-only or buffered-push (p or bp) as *push channels* and similarly to input channels that are either buffered-only or buffered-push as *buffered channels*.

All signals are assumed to have value types, i.e., any declaration $\kappa :_c A$ in Δ must have a value type A. The grammar for value types is given in Figure 2.

2.1 Clocks and $\textcircled{\exists}$

A clock is intuitively a set of push channels (p or bp), that the program may have to react to. For instance, $\emptyset$, {keyPressed} and {keyPressed, mouseCoord} are all examples of clocks for the example input context mentioned earlier. The type $\textcircled{\exists} A$ is a type of delayed computation on an existentially quantified clock. In other words, a value of type $\textcircled{\exists} A$ is a pair of a clock θ and a computation that will produce a value of type A once an update on one of the input channels in θ is received. We refer to such an update as a *tick* on the clock θ. For example, if the associated clock is {keyPressed, mouseCoord}, then the data of type A can be computed once keyPressed or mouseCoord receive new input.

$$\frac{}{\cdot \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad x \notin \mathrm{dom}\,(\Gamma)}{\Gamma, x : A \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_\Delta \theta : \mathsf{Clock} \qquad \Gamma \text{ tick-free}}{\Gamma, \checkmark_\theta \vdash_\Delta}$$

$$\frac{\Gamma \vdash_\Delta \theta : \mathsf{Clock} \qquad \Gamma \vdash_\Delta \theta' : \mathsf{Clock}}{\Gamma \vdash_\Delta \theta \sqcup \theta' : \mathsf{Clock}} \qquad \frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A}{\Gamma \vdash_\Delta \mathsf{cl}\,(v) : \mathsf{Clock}}$$

$$\frac{\Gamma' \text{ tick-free or } A \text{ stable} \qquad \Gamma, x : A, \Gamma' \vdash_\Delta}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A} \qquad \frac{}{\Gamma \vdash_\Delta \langle\rangle : 1} \qquad \frac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \mathsf{let}\, x = s \,\mathsf{in}\, t : B}$$

$$\frac{\Gamma, x : A \vdash_\Delta t : B \qquad \Gamma \text{ tick-free}}{\Gamma \vdash_\Delta \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash_\Delta t : A \to B \qquad \Gamma \vdash_\Delta t' : A}{\Gamma \vdash_\Delta t\, t' : B} \qquad \frac{\Gamma \vdash_\Delta t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \mathsf{in}_i\, t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash_\Delta t_i : B \qquad \Gamma \vdash_\Delta t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \mathsf{case}\, t \,\mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B} \qquad \frac{\Gamma \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta t' : B}{\Gamma \vdash_\Delta (t, t') : A \times B}$$

$$\frac{\Gamma \vdash_\Delta t : A_1 \times A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \pi_i\, t : A_i} \qquad \frac{\Gamma, \checkmark_\theta \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta \theta : \mathsf{Clock}}{\Gamma \vdash_\Delta \mathsf{delay}_\theta\, t : \textcircled{\exists} A} \qquad \frac{}{\Gamma \vdash_\Delta \mathsf{never} : \textcircled{\exists} A}$$

$$\frac{\kappa :_c A \in \Delta \qquad c \in \{\mathsf{p}, \mathsf{bp}\}}{\Gamma \vdash_\Delta \mathsf{wait}_\kappa : \textcircled{\exists} A} \qquad \frac{\kappa :_c A \in \Delta \qquad c \in \{\mathsf{b}, \mathsf{bp}\}}{\Gamma \vdash_\Delta \mathsf{read}_\kappa : A} \qquad \frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A \qquad \Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta \mathsf{adv}\, v : A}$$

$$\frac{\Gamma \vdash_\Delta v_1 : \textcircled{\exists} A_1 \qquad \Gamma \vdash_\Delta v_2 : \textcircled{\exists} A_2 \qquad \vdash \theta_1 \sqcup \theta_2 = \mathsf{cl}\,(v_1) \sqcup \mathsf{cl}\,(v_2) \qquad \Gamma, \checkmark_{\theta_1 \sqcup \theta_2}, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_{\theta_1 \sqcup \theta_2}, \Gamma' \vdash_\Delta \mathsf{select}\, v_1\, v_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)}$$

$$\frac{}{\Gamma \vdash_\Delta 0 : \mathsf{Nat}} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Nat}}{\Gamma \vdash_\Delta \mathsf{suc}\, t : \mathsf{Nat}} \qquad \frac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : \mathsf{Nat}, y : A \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta n : \mathsf{Nat}}{\Gamma \vdash_\Delta \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, n) : A}$$

$$\frac{\Gamma^\square, x : \textcircled{\forall} A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}\, x.t : A} \qquad \frac{\Gamma \vdash_\Delta x : \textcircled{\forall} A \qquad \Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta \mathsf{adv}\, x : A} \qquad \frac{\Gamma^\square \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{box}\, t : \square A}$$

$$\frac{\Gamma \vdash_\Delta t : \square A}{\Gamma \vdash_\Delta \mathsf{unbox}\, t : A} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Fix}\, \alpha.A}{\Gamma \vdash_\Delta \mathsf{out}\, t : A[\textcircled{\exists}(\mathsf{Fix}\, \alpha.A)/\alpha]} \qquad \frac{\Gamma \vdash_\Delta t : A[\textcircled{\exists}(\mathsf{Fix}\, \alpha.A)/\alpha]}{\Gamma \vdash_\Delta \mathsf{into}\, t : \mathsf{Fix}\, \alpha.A}$$

$$\cdot^\square = \cdot \qquad (\Gamma, \checkmark_\theta)^\square = \Gamma^\square \qquad (\Gamma, x : A)^\square = \begin{cases} \Gamma^\square, x : A & \text{if } A \text{ stable} \\ \Gamma^\square & \text{otherwise} \end{cases}$$

Fig. 3. Typing rules.

Since $\textcircled{\exists} A$ are existential types, one can obtain the clock $\mathsf{cl}\,(v)$ for any *value* of these types. The values of type $\textcircled{\exists} A$ are variables and wait_κ where κ is one of the push channels. The latter acts as a reference to the next value pushed on κ, and so intuitively $\mathsf{cl}\,(\mathsf{wait}_\kappa) = \{\kappa\}$. Clocks can also be combined using a union operator $\sqcup$. We also include an element never which is associated with the empty clock.

We use Fitch-style [Clouston 2018], rather than the more traditional dual context style [Davies and Pfenning 2001] for programming with the modal type constructors of Async RaTT. In the case of $\Circledcirc$, this means that introduction and elimination rules use a special symbol $\checkmark_\theta$, referred to as a *tick*, in the context. One can think of a tick $\checkmark_\theta$ as representing ticks of the clock θ, and it divides the judgement into variables (to the left of $\checkmark_\theta$) received before the tick, and everything else, which happens after the tick. For example, the elimination rule should be read as: If v has type $\Circledcirc A$ now, then after a tick on the clock $\mathsf{cl}\,(v)$, $\mathsf{adv}(v)$ has type A. Similarly, the introduction rule for $\Circledcirc$ should be read as: If t has type A after a tick on clock θ then $\mathsf{delay}_\theta\, t$ has type $\Circledcirc A$ now. Note that there can be at most one tick in a context. This is a restriction that is required for the proof of the productivity theorem (Theorem 4.1), and also appears in other languages in the RaTT family [Bahr et al. 2019, 2021]. However, Bahr [2022] shows that this restriction can be lifted by a program transformation that transforms a program typable with multiple ticks into one with only one tick and where adv is only applied to variables.

Operationally, the term $\mathsf{delay}_\theta\, t$ creates a delayed computation which is stored in a heap until the input data necessary for evaluating it is available. It is therefore not considered a value, rather $\mathsf{delay}_\theta\, t$ evaluates to a heap reference l to the delayed computation. Although heap references are part of Async RaTT, and even considered values (Figure 2), programmers are not allowed to use these directly, and there are therefore no typing rules for them.

Two delayed values $v_1 : \Circledcirc A_1$ and $v_2 : \Circledcirc A_2$ can be synchronised using select once a tick on the union clock $\mathsf{cl}\,(v_1) \sqcup \mathsf{cl}\,(v_2)$ has been received. The type of select $v_1\, v_2$ reflects the three possible cases for such a tick: It could be in one $\mathsf{cl}\,(v_i)$, but not the other, or it could be in both. For example, if the input is in $\mathsf{cl}\,(v_1)$, but not $\mathsf{cl}\,(v_2)$, then data of type $A_1 \times \Circledcirc A_2$ can be computed. The sync operator shown in section 1.1 can be defined using select. The idea of using a term like select to distinguish between these cases is due to Graulund et al. [2021], who only require two cases to be defined, resorting to non-deterministic choice in the case where the tick is in the intersection of the clocks. In Async RaTT, providing all three cases is crucial for the operational results of section 4.

Note that the rules for select and adv restrict the application of these constructions to values. One reason for this is that it simplifies the metatheory by preventing arbitrary terms occurring in contexts through clocks. It also means that clock expressions always are values that do not need to be evaluated. For example, evaluating $\mathsf{delay}_{\mathsf{cl}(t)}(\mathsf{adv}(t))$ requires evaluating t twice: first for evaluating the clock, and then to evaluate the term itself. Elimination of $\Circledcirc$ can be done for more general terms t using a combination of let-binding and adv.

2.2 Stable Types and Fixed Points

General values in Async RaTT can contain references to time-dependent data, such as delayed computations stored in the heap. One of the main purposes of the type system is to prevent such references to be dereferenced at times in the future when a delayed computation has been deleted from the heap. For this reason, arbitrary data should not be kept across time steps, and this is reflected in the type system in the variable introduction rule which prevents general variables to be introduced across ticks.

For some types, however, values can not contain such references. We refer to these as *stable types* and the grammar for these is given in Figure 2. Stable types include all those of the form $\Box A$, which classify computations that produce values of type A without any access to delayed computations. The introduction rule for $\Box$ constructs a delayed computation $\mathsf{box}(t)$ that can be evaluated at any time in the future. This requires t to be typed in a stable context, and so the hypothesis of the typing rule removes all ticks and all variables not of stable type from the context. The $\Box$ modality has a counit and a comultiplication $\Box \to \Box\Box$. Note that wait_κ and read_κ are stable in the sense

that $\vdash_\Delta \mathsf{box}(\mathsf{wait}_\kappa) : \Box(\textcircled{\exists} A)$ for any $\kappa :_c A \in \Delta$ where $c \in \{\mathsf{p}, \mathsf{bp}\}$ and $\vdash_\Delta \mathsf{box}(\mathsf{read}_\kappa) : \Box A$ for any $\kappa :_c A \in \Delta$ where $c \in \{\mathsf{b}, \mathsf{bp}\}$.

Async RaTT is a terminating calculus in the sense that each step of computation terminates. It does, however, still allow recursive definitions through a fixed point operator, whose type ensures that recursive calls are only done in later time steps. More precisely, the recursion variable x in $\mathsf{fix}\, x.t$ has type $\textcircled{\forall} A$, which means that the recursive definition can be unfolded to produce a term of type A any time in the future, but not now. This is ensured through the elimination rule for $\textcircled{\forall}$ which allows it to be advanced using a tick on any clock typable in the current context. Since fixed points can be called recursively at any time in the future, these must be stable, and so t is required to be typable in a stable context.

The types $\mathsf{Fix}\, \alpha.A$ are guarded recursive types that unfold to $A[\textcircled{\exists}(\mathsf{Fix}\, \alpha.A)/\alpha]$ via the terms into and out. The most important of these types is $\mathsf{Sig}\, A$ defined as $\mathsf{Fix}\, \alpha.(A \times \alpha)$, which unfolds to $A \times \textcircled{\exists}(\mathsf{Sig}\, A)$. A signal consists of a current value and a delayed tail, which at some time in the future may return a new signal. Any push channel $\kappa :_c A \in \Delta$, where $c \in \{\mathsf{p}, \mathsf{bp}\}$, induces a stable signal:

$$\mathsf{box}\left(\mathsf{fix}\, x.\mathsf{delay}_{\mathsf{cl}(\mathsf{wait}_\kappa)}\ (\mathsf{into}\ (\mathsf{adv}(\mathsf{wait}_\kappa), \mathsf{adv}(x)))\right) : \Box(\textcircled{\exists}(\mathsf{Sig}\, A))$$

where the recursion variable x has type $\textcircled{\forall}(\textcircled{\exists} A)$. These signals, of course, operate on a fixed clock $\{\kappa\}$, but in general, the clock associated with the tail of a signal may change from one step to the next, which we shall see examples of in section 3.

Besides all these constructions, Async RaTT also has a number of standard constructions from functional programming: sum types, product types, natural numbers and function types. The typing rules for these are completely standard, with the exception that function types can only be constructed in contexts with no ticks. Similar restrictions are known from other calculi in the RaTT family [Bahr et al. 2019, 2021], and are necessary for the results of section 4. The aforementioned program transformation by Bahr [2022] also removes this restriction. Note that function types are not stable, since time-dependent references can be stored in closures.

3 PROGRAMMING IN ASYNC RATT

In this section, we demonstrate the expressiveness of Async RaTT with a number of examples. To this end, we assume a surface language that extends Async RaTT with syntactic sugar for pattern matching, recursion, and top-level definitions. These can be easily elaborated into the Async RaTT calculus as described in section 3.5.

3.1 Simple Signal Combinators

We start by implementing a small set of simple combinators to manipulate signals, i.e., elements of the guarded recursive type $\mathsf{Sig}\, A$ defined as $\mathsf{Fix}\, \alpha.(A \times \alpha)$. For readability we use the shorthand $s :: t$ for $\mathsf{into}\, (s, t)$, such that, given $s : A$ and $t : \textcircled{\exists}(\mathsf{Sig}\, A)$, we have that $s :: t : \mathsf{Sig}\, A$.

We start with perhaps the simplest signal combinator:

$$\begin{aligned} &\mathit{map} : \Box\ (A \to B) \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\ &\mathit{map}\ f\ (x :: xs) = \mathsf{unbox}\ f\ x :: \mathsf{delay}\ (\mathit{map}\ f\ (\mathsf{adv}\ xs)) \end{aligned}$$

The *map* combinator takes a stable function f and applies it pointwise to a given signal. The fact that f is of type $\Box(A \to B)$ rather than just $A \to B$ is crucial: Since $A \to B$ is not a stable type, f would otherwise not be in scope under the delay, where we need f for the recursive call. It also has an intuitive justification: The function will be applied to values of the input signal arbitrarily far into the future, but a closure of type $A \to B$ may contain references to delayed computations that may have been garbage collected in the future.

The *map* combinator is stateless in the sense that the current value of the output signal only depends on the current value of the input signal. We can generalise this combinator to *scan*, which produces an output signal that in addition may depend on the previous value of the output signal:

$$
\begin{aligned}
&\mathit{scan} : \mathsf{stable}\ B \Rightarrow \Box\ (B \to A \to B) \to B \to \mathsf{Sig}\ A \to \mathsf{Sig}\ B \\
&\mathit{scan}\ f\ \mathit{acc}\ (a :: \mathit{as}) = \mathit{acc}' :: \mathsf{delay}\ (\mathit{scan}\ f\ \mathit{acc}'\ (\mathsf{adv}\ \mathit{as})) \\
&\quad \textbf{where}\ \mathit{acc}' = \mathsf{unbox}\ f\ \mathit{acc}\ a
\end{aligned}
$$

Every time the input signal updates, the output signal produces a new value based on the current value of the input signal and the previous value of the output signal. Since the previous value of the output signal is accessed, B must be a stable type. We use the $\Rightarrow$ notation to delineate such constraints from the type signature.

For example, we can use *scan* to produce the sum of an input signal of numbers:

$$
\begin{aligned}
&\mathit{sum} : \mathsf{Sig}\ \mathsf{Nat} \to \mathsf{Sig}\ \mathsf{Nat} \\
&\mathit{sum} = \mathit{scan}\ (\mathsf{box}\ (\lambda m\ n \to m + n))\ 0
\end{aligned}
$$

Often we only have access to a *delayed* signal. For instance, for each push channel $\kappa :_c A \in \Delta$, $c \in \{\mathsf{p}, \mathsf{bp}\}$ we have the signal

$$
\begin{aligned}
&\mathit{sigAwait}_\kappa : \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \\
&\mathit{sigAwait}_\kappa = \mathsf{delay}\ (\mathsf{adv}\ \mathsf{wait}_\kappa :: \mathit{sigAwait}_\kappa)
\end{aligned}
$$

For example, we might have the push-only channels $\mathsf{mouseClick} :_\mathsf{p} 1$ or $\mathsf{keyPress} :_\mathsf{p} \mathsf{KeyCode}$ available. We can derive a version of *scan* for such signals:

$$
\begin{aligned}
&\mathit{scanAwait} : \mathsf{stable}\ B \Rightarrow \Box\ (B \to A \to B) \to B \to \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ B \\
&\mathit{scanAwait}\ f\ \mathit{acc}\ \mathit{as} = \mathit{acc} :: \mathsf{delay}\ (\mathit{scan}\ f\ \mathit{acc}\ (\mathsf{adv}\ \mathit{as}))
\end{aligned}
$$

A simple use case of *scanAwait* is a combinator that counts the updates of a given delayed signal, e.g., the number of key presses:

$$
\begin{aligned}
&\mathit{count} : \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \to \mathsf{Nat} \to \mathsf{Sig}\ \mathsf{Nat} \\
&\mathit{count}\ s\ n = \mathit{scanAwait}\ (\mathsf{box}\ (\lambda \_\ m \to m + 1))\ n\ s
\end{aligned}
$$

Finally, we have the most simple combinator that simply produces a constant signal:

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

In isolation this combinator may appear to be of little use. Its utility becomes apparent once we also have the switching combinators introduced in the next section.

3.2 Concurrent Signal Combinators

The combinators we looked at so far only consumed a single signal, and thus had no need to account for the concurrent behaviour of two or more clocks. For example, we may have two input signals produced by two redundant sensors that independently provide a reading we are interested in. To combine these two signals, we can interleave them using the following combinator:

$$
\begin{aligned}
&\mathit{interleave} : \Box\ (A \to A \to A) \to \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \to \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \to \Diamond\!\!\!-\ (\mathsf{Sig}\ A) \\
&\mathit{interleave}\ f\ \mathit{xs}\ \mathit{ys} = \mathsf{delay}\ (\textbf{case}\ \mathsf{select}\ \mathit{xs}\ \mathit{ys}\ \textbf{of} \\
&\qquad\qquad \mathit{Left}\ (x :: \mathit{xs}')\ \ \ \mathit{ys}'\ .x :: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{ys}' \\
&\qquad\qquad \mathit{Right}\ \ \ \mathit{xs}'\ \ (y :: \mathit{ys}').y :: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{ys}' \\
&\qquad\qquad \mathit{Both}\ (x :: \mathit{xs}')\ (y :: \mathit{ys}').\mathsf{unbox}\ f\ x\ y :: \mathit{interleave}\ f\ \mathit{xs}'\ \mathit{xs}')
\end{aligned}
$$

In this and subsequent definitions, we use the shorthands *Left*, *Right*, and *Both*, in the expected way. For example, *Left s t* is short for $\mathsf{in}_1(\mathsf{in}_1(s,t))$, i.e., the case that the left clock ticked first. The *interleave* combinator uses select in order to wait until at least one of the input signals ticks, and then updates the output signal accordingly. In case that both signals tick simultaneously, the provided merging function f is applied. For example, f could just always use the value of the first signal or take the average. Note that the produced signal combines the clocks of the input signals, i.e., it ticks whenever either of the input signals ticks.

We might also be interested in the values of both input signals simultaneously, in which case we would use *zip*:

$$
\begin{aligned}
&\mathit{zip} : \mathsf{stable}\ A, B \Rightarrow \mathsf{Sig}\ A \to \mathsf{Sig}\ B \to \mathsf{Sig}\ (A \times B)\\
&\mathit{zip}\ (x :: xs)\ (y :: ys) = (x, y) :: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ ys\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Left}\quad xs'\ ys'.\mathit{zip}\ xs'\ (y :: ys')\\
&\qquad\qquad\qquad\qquad \mathit{Right}\ xs'\ ys'.\mathit{zip}\ (x :: xs')\ ys'\\
&\qquad\qquad\qquad\qquad \mathit{Both}\quad xs'\ ys'.\mathit{zip}\ xs'\ ys')
\end{aligned}
$$

Similarly to *interleave*, the output signal produced by *zip* ticks whenever either of the input signals does. However, note that in the *Left* and *Right* cases, we copy the previously observed value from the signal that did not tick into the future. Hence, we need both types, A and B, to be stable.

Finally, we consider the switching of signals. We wish to produce a signal that behaves initially like a given input signal, but switches to a different signal as soon as some event happens. This idea is implemented in the *switch* function:

$$
\begin{aligned}
&\mathit{switch} : \mathsf{Sig}\ A \to \Im\ (\mathsf{Sig}\ A) \to \mathsf{Sig}\ A\\
&\mathit{switch}\ (x :: xs)\ d = x :: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ d\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Left}\quad xs'\ d'.\mathit{switch}\ xs'\ d'\\
&\qquad\qquad\qquad\qquad \mathit{Right}\ \_\quad d'.d'\\
&\qquad\qquad\qquad\qquad \mathit{Both}\quad \_\quad d'.d')
\end{aligned}
$$

The event that represents the future change of the signal is represented as a delayed signal, and as soon as this delayed signal ticks, as in the *Right* and *Both* cases, it takes over. With the help of *switch* we can construct dynamic dataflow graphs since we replace a given signal with an entirely new signal, which may depend on different input channels and intermediate signals compared to the original signal.

We will demonstrate an example of this dynamic behaviour in the next section. In preparation for that we devise a variant of *switch*, where the new signal depends on the value of the previous signal:

$$
\begin{aligned}
&\mathit{switchf} : \mathsf{stable}\ A \Rightarrow \mathsf{Sig}\ A \to \Im\ (A \to \mathsf{Sig}\ A) \to \mathsf{Sig}\ A\\
&\mathit{switchf}\ (x :: xs)\ d = x :: \mathsf{delay}\ (\mathbf{case}\ \mathsf{select}\ xs\ d\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad \mathit{Left}\qquad xs'\ \ d'.\mathit{switchf}\ xs'\ d'\\
&\qquad\qquad\qquad\qquad \mathit{Right}\qquad \_\ \ d'.d'\ x\\
&\qquad\qquad\qquad\qquad \mathit{Both}\ (x' :: xs')\ d'.d'\ x')
\end{aligned}
$$

Instead of a new signal, this combinator waits for a *function* that produces the new signal, and we feed this function the last value of the first signal.

3.3 A Simple GUI Example

To demonstrate how to use our signal combinators, we consider a very simple example of a GUI application: Our goal is to write a reactive program with two output channels that describe the

contents of two text fields. To this end, the two output channels are given the type Sig Nat. The number displayed in text fields should be incremented each time the user clicks a button, which is available as an input channel $\mathsf{up} :_{\mathsf{p}} 1 \in \Delta$. However, there is only one 'up' button and the user can change which text field should be changed by the 'up' button using a 'toggle' button, which is available as an input channel $\mathsf{toggle} :_{\mathsf{p}} 1 \in \Delta$.

That means, the contents of the first text field can be described by the *count* combinator, but then switches to the signal described by the *const* combinator when 'toggle' is pressed. The behaviour of the other text field is reversed: first *const*, then *count*. This continuous toggling between behaviours can be concisely described by the following combinator:

$$\begin{aligned}
&\mathit{toggleSig} : \mathsf{stable}\ A \Rightarrow \Box\ (\textcircled{\exists}\ 1) \to \Box\ (A \to \mathsf{Sig}\ A) \to \Box\ (A \to \mathsf{Sig}\ A) \to A \to \mathsf{Sig}\ A \\
&\mathit{toggleSig}\ \mathit{tog}\ f\ g\ x = \mathit{switchf}\ (\mathsf{unbox}\ f\ x)\ (\mathsf{delay}\ (\mathsf{adv}\ \mathit{tick}; \mathit{toggleSig}\ \mathit{tog}\ g\ f)) \\
&\quad \textbf{where}\ \mathit{tick} = \mathsf{unbox}\ \mathit{tog}
\end{aligned}$$

The first argument provides the events that determine when to toggle between the two behaviours, which in turn are given as the next two arguments. In the implementation we use the notation $s; t$ as a shorthand for $\mathsf{let}\ \langle\rangle = s\ \mathsf{in}\ t$. The *toggleSig* combinator uses *switchf* to start with the first signal provided by f, but then switches to g as soon as the toggle *tog* ticks by using a recursive call that swaps the order of the two arguments f and g.

The output channels that describe the two text fields can now be implemented by providing the appropriate input signals to *toggleSig*:

$$\begin{aligned}
&\mathit{field1}, \mathit{field2} : \mathsf{Sig}\ \mathsf{Nat} \\
&\mathit{field1} = \mathit{toggleSig}\ (\mathsf{box}\ \mathsf{wait}_{\mathsf{toggle}})\ (\mathsf{box}\ (\mathit{count}\ \mathit{sigAwait}_{\mathsf{up}}))\ (\mathsf{box}\ \mathit{const}) \qquad 0 \\
&\mathit{field2} = \mathit{toggleSig}\ (\mathsf{box}\ \mathsf{wait}_{\mathsf{toggle}})\ (\mathsf{box}\ \mathit{const}) \qquad (\mathsf{box}\ (\mathit{count}\ \mathit{sigAwait}_{\mathsf{up}}))\ 0
\end{aligned}$$

Note that the dataflow graph changes during the execution of the program and how that change is reflected in the clock associated with the output channels: the output channel for the first text field first has the clock $\{\mathsf{up}, \mathsf{toggle}\}$ as it must both count the number of times the 'up' button is clicked and change its behaviour in reaction to the 'toggle' button being clicked. Once the 'toggle' button has been clicked, the clock for output channel for the text field changes to $\{\mathsf{toggle}\}$ as it now ignores the 'up' button. We will examine the run-time behaviour of this example in more detail in section 4.3.

3.4 Integral and Derivative

Buffered input channels can be used to represent input signals that change at discrete points in time, but whose current value can be accessed at any time. For example, given a buffered push channel $\kappa :_{\mathsf{bp}} A \in \Delta$, we can construct the following signal (using $\mathit{sigAwait}_\kappa$ from section 3.1):

$$\begin{aligned}
&\mathit{sig}_\kappa : \mathsf{Sig}\ A \\
&\mathit{sig}_\kappa = \mathsf{read}_\kappa :: \mathit{sigAwait}_\kappa
\end{aligned}$$

To illustrate what we can do with such input signals, we assume that Async RaTT has a stable type Float together with typical operations on floating-point numbers. Figure 4 gives the definition of two signal combinators that each take a floating-point-valued signal and produce the integral and the derivative of that signal. To this end, we assume a buffered push channel $\mathsf{sample} :_{\mathsf{bp}} \mathsf{Float} \in \Delta$ that produces a new floating-point number s at some fixed interval (e.g., 10 times per second). This number s is the number of seconds since the last update on the channel, e.g., $s = 0.1$ if sample ticks 10 times per second.

The *integral* combinator produces the integral of a given signal starting from a given constant that is provided as the first argument. Its implementation uses a simple approximation that samples

```
integral : Float → Sig Float → Sig Float
integral cur (0 :: xs) = cur :: delay (integral cur (adv xs))
integral cur (x :: xs) = cur :: delay (case select xs wait_sample of
                                         Left xs' _.        integral cur xs'
                                         Right xs' dt.      integral (cur + x × dt) (x :: xs')
                                         Both (x' :: xs') dt.integral (cur + x' × dt) (x' :: xs'))

derivative : Sig Float → Sig Float
derivative xs = der 0 (head xs) xs where
  der : Float → Float → Sig Float → Sig Float
  der 0 last (x :: xs) = 0 :: delay (let x' :: xs' = adv xs
                                     in der ((x' − x) / read_sample) x (x' :: xs'))
  der d last (x :: xs) = d :: delay (case select xs wait_sample of
                                       Left xs' _.        der d last xs'
                                       Right xs' dt.      der ((x − last) / dt) x (x :: xs')
                                       Both (x' :: xs') dt.der ((x' − last) / dt) x' (x' :: xs'))
```

Fig. 4. Integral and derivative signal combinators.

the value of the underlying signal each time the sample channel produces a value and adds the area of the rectangle formed by the value of the signal and the time that has passed since the last sampling.

The first equation of the definition is an optimisation and could be omitted. It says that if the current value of the underlying signal is 0, we simply wait until the underlying signal is updated, since the value of the integral won't change until the underlying signal has a non-zero value. Hence, we don't have to sample every time the sample channel ticks.

Similarly to *integral*, we can implement a function *derivative* that, given a floating-point-valued signal, produces its derivative. Like the *integral* function, also *derivative* samples the underlying signal every time sample ticks. To do so it uses the auxiliary function *der*, which takes two additional arguments: the current value of the derivative and the value of the underlying signal at the time of the most recent input from of the sample channel. Similarly to *integral*, the first line of *der* performs an optimisation: If the computed value of the derivative is 0, the sampling will pause until the underlying signal is updated. As soon as it does, we pretend that sample ticked to provide a timely update of the derivative.

These two combinators can be easily generalised from floating-point values to any vector space. This can then be used to describe complex behaviours in reaction to multidimensional sensor data.

3.5 Elaboration of Surface Syntax into Core Calculus

To illustrate how the surface language elaborates into the Async RaTT core calculus, reconsider the definition of *map*

```
map : □ (A → B) → Sig A → Sig B
map f (x :: xs) = unbox f x :: delay (map f (adv xs))
```

which elaborates to the following term in plain Async RaTT:

$$map = \mathsf{fix}\, r.\lambda f.\lambda s.\mathsf{let}\, x = \pi_1(\mathsf{out}\, s)\, \mathsf{in}\, \mathsf{let}\, xs = \pi_2(\mathsf{out}\, s) \\ \mathsf{in}\, \mathsf{into}\Big(unbox\ f\ x, \mathsf{delay}_{\mathsf{cl}(xs)}\,(\mathsf{adv}\, r\ f\ (\mathsf{adv}\, xs))\Big)$$

Recall that $s :: t$ is a shorthand for $\mathsf{into}\,(s, t)$. Pattern matching is translated into the corresponding elimination forms, out for recursive types, π_i for product types, and case for sum types. The recursion syntax – *map* occurs in the body of its definition – is translated to a fixed point $\mathsf{fix}\, r.t$ so that the recursive occurrence of *map* is replaced by $\mathsf{adv}\, r$. Hence, recursive calls must always occur in the scope of a ✓, which is the case in the definition of *map* as it appears in the scope of a delay. Moreover, we elide the subscript $\mathsf{cl}\,(xs)$ of delay since it can be uniquely inferred from the fact that we have the term $\mathsf{adv}\, xs$ in the scope of the delay.

In addition, we make use of top-level definitions like *map* and *scan*, which may be used in any context later on. For example, *scan* is used in the definition of *scanAwait* in the scope of a ✓. We can think these top-level definitions to be implicitly boxed when defined and unboxed when used later on. That is, these definitions are translated as follows to the core calculus:

$$\mathsf{let}\ scan = \mathsf{box}(\dots)\ \mathsf{in} \\ \mathsf{let}\ scanAwait = \mathsf{box}(\dots(\mathsf{unbox}\ scan)\dots))\ \mathsf{in} \\ \dots$$

4 OPERATIONAL SEMANTICS AND OPERATIONAL GUARANTEES

We describe the operational semantics of Async RaTT in two stages: We begin in section 4.1 with the *evaluation semantics* that describes how Async RaTT terms are evaluated at a particular point in time. Among other things, the evaluation semantics describes the computation that must happen to make updates in reaction to the arrival of new input on a push channel. We then describe in section 4.2 the *reactive semantics* that captures the dynamic behaviour of Async RaTT programs over time. The reactive semantics is a machine that waits for new input to arrive, and then computes new values for output channels that depend on the newly arrived input. For the latter, the reactive semantics invokes the evaluation semantics to perform the necessary updating computations.

Finally, after demonstrating the operational semantics on an example in section 4.3, we conclude the discussion of the operational semantics in section 4.4 with a precise account of our main technical results about the properties of the operational semantics: productivity, causality, signal independence, and the absence of implicit space leaks. To prove the latter, the evaluation semantics uses a store in which both external inputs and delayed computations are stored. Delayed computations are garbage collected as soon as the data on which they depend has arrived. In this fashion, Async RaTT avoids implicit space leaks by construction, provided we can prove that the operational semantics never gets stuck.

4.1 Evaluation Semantics

Figure 5 defines the evaluation semantics as a deterministic big-step operational semantics. We write $\langle t; \sigma\rangle \Downarrow^{\iota} \langle v; \tau\rangle$ to denote that when given a term t, a *store* σ, and an *input buffer* ι, the machine computes a value v and a new store τ. During the computation, the machine may defer computations into the future by storing unevaluated terms in the store σ to be retrieved and evaluated later. Conversely, the machine may also retrieve terms whose evaluation have been deferred at an earlier time and evaluate them now. In addition, the machine may read the new value of the most recently updated push channel from the store σ and read the current value of any buffered channel from the input buffer ι.

$$\frac{}{\langle v;\sigma\rangle \Downarrow^\iota \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow^\iota \langle v';\sigma''\rangle}{\langle (t,t');\sigma\rangle \Downarrow^\iota \langle (v,v');\sigma''\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^\iota \langle (v_1,v_2);\sigma'\rangle \qquad i\in\{1,2\}}{\langle \pi_i(t);\sigma\rangle \Downarrow^\iota \langle v_i;\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \mathsf{in}_i(t);\sigma\rangle \Downarrow^\iota \langle \mathsf{in}_i(v);\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^\iota \langle \mathsf{in}_i(v);\sigma'\rangle \qquad \langle t_i[v/x];\sigma'\rangle \Downarrow^\iota \langle v_i;\sigma''\rangle \qquad i\in\{1,2\}}{\langle \mathsf{case}\ t\ \mathsf{of}\ \mathsf{in}_1\ x.t_1;\mathsf{in}_2\ x.t_2;\sigma\rangle \Downarrow^\iota \langle v_i;\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow^\iota \langle \lambda x.s;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle \qquad \langle s[v/x];\sigma''\rangle \Downarrow^\iota \langle v';\sigma'''\rangle}{\langle t\,t';\sigma\rangle \Downarrow^\iota \langle v';\sigma'''\rangle}$$

$$\frac{\langle s;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle \qquad \langle t[v/x];\sigma'\rangle \Downarrow^\iota \langle w;\sigma''\rangle}{\langle \mathsf{let}\ x=s\ \mathsf{in}\ t;\sigma\rangle \Downarrow^\iota \langle w;\sigma''\rangle} \qquad \frac{\kappa\in \mathsf{dom}\,(\iota)}{\langle \mathsf{read}_\kappa;\sigma\rangle \Downarrow^\iota \langle \iota(\kappa);\sigma\rangle}$$

$$\frac{l=\mathsf{alloc}^{|\theta|}\,(\sigma)}{\langle \mathsf{delay}_\theta\ t;\sigma\rangle \Downarrow^\iota \langle l;(\sigma,l\mapsto t)\rangle} \qquad \frac{l=\mathsf{alloc}^{\emptyset}\,(\sigma)}{\langle \mathsf{never};\sigma\rangle \Downarrow^\iota \langle l;\sigma\rangle}$$

$$\frac{}{\langle \mathsf{adv}\ \mathsf{wait}_\kappa;\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L\rangle \Downarrow^\iota \langle v;\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L\rangle} \qquad \frac{\langle \eta_N(l);\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L\rangle \Downarrow^\iota \langle w;\sigma\rangle}{\langle \mathsf{adv}\ l;\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L\rangle \Downarrow^\iota \langle w;\sigma\rangle}$$

$$\frac{\kappa\in |\mathsf{cl}\,(v_i)|\setminus|\mathsf{cl}\,(v_{3-i})| \qquad \langle \mathsf{adv}\ v_i;\eta_N\,\langle\kappa\mapsto w\rangle\,\eta_L\rangle \Downarrow^\iota \langle u_i;\sigma\rangle \qquad u_{3-i}=v_{3-i}}{\langle \mathsf{select}\ v_1\ v_2;\eta_N\,\langle\kappa\mapsto w\rangle\,\eta_L\rangle \Downarrow^\iota \langle \mathsf{in}_1(\mathsf{in}_i\,(u_1,u_2));\sigma\rangle}$$

$$\frac{\kappa\in |\mathsf{cl}\,(v_1)|\cap|\mathsf{cl}\,(v_2)| \qquad \langle \mathsf{adv}\ v_1;\eta_N\,\langle\kappa\mapsto w\rangle\,\eta_L\rangle \Downarrow^\iota \langle u_1;\sigma\rangle \qquad \langle \mathsf{adv}\ v_2;\sigma\rangle \Downarrow^\iota \langle u_2;\sigma'\rangle}{\langle \mathsf{select}\ v_1\ v_2;\eta_N\,\langle\kappa\mapsto w\rangle\,\eta_L\rangle \Downarrow^\iota \langle \mathsf{in}_2\,(u_1,u_2)\,;\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}{\langle \mathsf{suc}\ t;\sigma\rangle \Downarrow^\iota \langle \mathsf{suc}\ v;\sigma'\rangle} \qquad \frac{\langle n;\sigma\rangle \Downarrow^\iota \langle 0;\sigma'\rangle \qquad \langle s;\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s,x\,y.t,n);\sigma\rangle \Downarrow^\iota \langle v;\sigma''\rangle}$$

$$\frac{\langle n;\sigma\rangle \Downarrow^\iota \langle \mathsf{suc}\ v;\sigma'\rangle \qquad \langle \mathsf{rec}_{\mathsf{Nat}}(s,x\,y.t,v);\sigma'\rangle \Downarrow^\iota \langle v';\sigma''\rangle \qquad \langle t[v/x,v'/y];\sigma''\rangle \Downarrow^\iota \langle w;\sigma'''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s,x\,y.t,n);\sigma\rangle \Downarrow^\iota \langle w;\sigma'''\rangle}$$

$$\frac{\langle t[\mathsf{dfix}\,x.t/x];\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}{\langle \mathsf{adv}\,(\mathsf{dfix}\,x.t);\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle} \qquad \frac{\langle t[\mathsf{dfix}\,x.t/x];\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}{\langle \mathsf{fix}\,x.t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow^\iota \langle \mathsf{box}\ t';\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle}{\langle \mathsf{unbox}\ t;\sigma\rangle \Downarrow^\iota \langle v;\sigma''\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}{\langle \mathsf{into}\ t;\sigma\rangle \Downarrow^\iota \langle \mathsf{into}\ v;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^\iota \langle \mathsf{into}\ v;\sigma'\rangle}{\langle \mathsf{out}\ t;\sigma\rangle \Downarrow^\iota \langle v;\sigma'\rangle}$$

Fig. 5. Operational semantics.

To facilitate the delay of computations, the syntax of the language features heap locations l, which are not typable in the calculus but may be introduced by the machine during evaluation. A heap location represents a delayed computation that can be resumed once a particular clock has ticked, which indicates that the data the delayed computation is waiting for has arrived. To this end, each heap location l is associated with a clock, denoted $\mathsf{cl}\,(l)$. As soon as the clock $\mathsf{cl}\,(l)$ ticks, the delayed computation represented by l can be resumed by retrieving the unevaluated term stored at heap location l and evaluating it. We write Loc for the set of all heap locations and assume

that for each clock Θ, there are countably infinitely many locations l with $\mathsf{cl}\,(l) = \Theta$. A clock Θ is a finite set of push channels drawn from $\mathsf{dom}\,(\Delta)$, and it ticks any time any of its channels $\kappa \in \Theta$ is updated. For example, assuming an input context Δ for a GUI, the clock $\{\mathsf{keyPressed}, \mathsf{mouseCoord}\}$ ticks whenever the user presses a key or moves the mouse. Note that we are now more precise in distinguishing *clock expressions*, typically denoted θ, and *clocks*, typically denoted Θ. A closed clock expression θ evaluates to a clock $|\theta|$ as follows:

$$|\mathsf{cl}\,(l)| = \mathsf{cl}\,(l) \qquad |\mathsf{cl}\,(\mathsf{wait}_\kappa)| = \{\kappa\} \qquad |\theta \sqcup \theta'| = |\theta| \cup |\theta'|$$

Delayed computations reside in a heap, which is simply a finite mapping η from heap locations to terms. Of particular interest are heaps η whose locations, denoted $\mathsf{dom}\,(\eta)$, each have a clock that contains a given input channel κ:

$$\mathsf{Heap}^\kappa = \{\eta \in \mathsf{Heap} \mid \forall l \in \mathsf{dom}\,(\eta)\,.\,\kappa \in \mathsf{cl}\,(l)\,\}$$

It is safe to evaluate terms stored in a heap $\eta \in \mathsf{Heap}^\kappa$ as soon as a new value on the input channel κ has arrived. This intuition is reflected in the representation of stores σ, which can be in one of two forms: a single-heap store η_L or a two-heap store $\eta_N\,\langle \kappa \mapsto v\rangle\,\eta_L$ with $\eta_N \in \mathsf{Heap}^\kappa$. We typically refer to η_L as the *later* heap, which is used to store delayed computations for later, and to η_N as the *now* heap, whose stored terms are safe to be evaluated now. The $\langle \kappa \mapsto v\rangle$ component of a two-heap store indicates that the input channel κ has been updated to the new value v. The machine can thus safely resume computations from η_N since the data that the delayed computations in η_N were waiting for has arrived.

Let's first consider the semantics for delay: To allocate fresh locations in the store, we assume a function $\mathsf{alloc}^\Theta\,(\cdot)$, which, if given a store η_L or $\eta_N\,\langle \kappa \mapsto v\rangle\,\eta_L$, produces a location $l \notin \mathsf{dom}\,(\eta_L)$ with $\mathsf{cl}\,(l) = \Theta$. This results in a store $\eta_L, l \mapsto t$ or $\eta_N\,\langle \kappa \mapsto v\rangle\,\eta_L, l \mapsto t$, respectively, where $\eta_L, l \mapsto t$ denotes the heap η_L extended with the mapping $l \mapsto t$.

Conversely, $\mathsf{adv}\,l$ retrieves a previously delayed computation. The typing discipline ensures that $\mathsf{adv}\,l$ will only be evaluated in the context of a store of the form $\eta_N\,\langle \kappa \mapsto v\rangle\,\eta_L$ with $l \in \mathsf{dom}\,(\eta_N)$ and therefore also $\kappa \in \mathsf{cl}\,(l)$. In addition, adv may be applied to wait_κ which simply looks up the new value v from the channel κ.

The select combinator allows us to interact with two delayed computations simultaneously. Its semantics checks for the three possible contingencies, namely which non-empty subset of the two delayed computations has been triggered. Each of the two argument values v_1 or v_2 is either a heap location or wait_κ and thus the machine can simply check whether the current input channel κ is in the clocks associated with v_1, v_2, or both. Depending on the outcome, the machine advances the corresponding value(s).

Finally, the fixed point combinator fix is evaluated with the help of the combinator dfix, which similarly to heap locations is not typable in the calculus but is introduced by the machine. Intuitively speaking, we can think of a value of the form $\mathsf{dfix}\,x.t$ as shorthand for $\Lambda\theta.(\mathsf{delay}_\theta(\mathsf{fix}\,x.t))$. That is, $\mathsf{dfix}\,x.t$ is a thunk that, when given a clock θ, produces a delayed computation on θ, which in turn evaluates a fixed point once θ ticks. The action of the adv combinator for $\textcircled{\scriptsize\textsf{v}}$ can thus also be interpreted as first providing the clock θ and then advancing the delayed computation $\mathsf{delay}_\theta(\mathsf{fix}\,x.t)$, which means evaluating $\mathsf{fix}\,x.t$.

4.2 Reactive Semantics

An Async RaTT program interacts with its environment by receiving input from a set of input channels and in return sends output to a set of output channels. The input context Δ describes the available input channels. In addition, we also have an output context Γ_{out}, that only contains

$$\textsc{init}\ \frac{\langle \langle t \rangle ; \emptyset \rangle \Downarrow^{\iota} \langle \langle v_1 :: l_1, \ldots, v_m :: l_m \rangle ; \eta \rangle}{\langle t; \iota \rangle \overset{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m}{\Longrightarrow} \langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m; \eta; \iota \rangle}$$

$$\textsc{input}\ \frac{\iota' = \iota[\kappa \mapsto v] \text{ if } \kappa \in \mathrm{dom}\,(\iota) \text{ otherwise } \iota' = \iota}{\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \left\langle N; [\eta]_{\kappa \in} \langle \kappa \mapsto v \rangle [\eta]_{\kappa \notin} ; \iota' \right\rangle}$$

$$\textsc{output-end}\ \frac{}{\langle \cdot; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{\cdot}{\Longrightarrow} \langle \cdot; \eta_L; \iota \rangle}$$

$$\textsc{output-skip}\ \frac{\kappa \notin \mathrm{cl}\,(l) \qquad \langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{O}{\Longrightarrow} \langle N'; \eta; \iota \rangle}{\langle x \mapsto l, N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{O}{\Longrightarrow} \langle x \mapsto l, N'; \eta; \iota \rangle}$$

$$\textsc{output-compute}\ \frac{\kappa \in \mathrm{cl}\,(l) \qquad \langle \mathrm{adv}\, l; \eta_N \langle \kappa \mapsto v \rangle \eta_L \rangle \Downarrow^{\iota} \langle v' :: l'; \sigma \rangle \qquad \langle N; \sigma; \iota \rangle \overset{O}{\Longrightarrow} \langle N'; \eta; \iota \rangle}{\langle x \mapsto l, N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{x \mapsto v', O}{\Longrightarrow} \langle x \mapsto l', N'; \eta; \iota \rangle}$$

Fig. 6. Reactive semantics.

variables $x : A$, where A is a value type. We refer to the variables in Γ_{out} as output channels. Taken together, we call the pair consisting of Δ and Γ_{out} a *reactive interface*, written $\Delta \Rightarrow \Gamma_{\mathrm{out}}$.

Given an output interface $\Gamma_{\mathrm{out}} = x_1 : A_1, \ldots, x_n : A_n$, we define the type $\mathrm{Prod}\,(\Gamma_{\mathrm{out}})$ as the product of all types in Γ_{out}, i.e., $\mathrm{Prod}\,(\Gamma_{\mathrm{out}}) = \mathrm{Sig}\, A_1 \times \cdots \times \mathrm{Sig}\, A_n$. The n-ary product type used here can be encoded using the binary product type and the unit type in the standard way. An Async RaTT term t is said to be a reactive program implementing the reactive interface $\Delta \Rightarrow \Gamma_{\mathrm{out}}$, denoted $t : \Delta \Rightarrow \Gamma_{\mathrm{out}}$, if $\vdash_{\Delta} t : \mathrm{Prod}\,(\Gamma_{\mathrm{out}})$.

The operational semantics of a reactive program is described by the machine in Figure 6. The state of the machine can be of two different forms: Initially, the machine is in a state of the form $\langle t; \iota \rangle$, where $t : \Delta \Rightarrow \Gamma_{\mathrm{out}}$ is the reactive program and ι is the initial input buffer, which contains the initial values of all buffered input channels. Subsequently, the machine state is a pair $\langle N; \sigma; \iota \rangle$, where N is a sequence of the form $x_1 \mapsto l_1, \ldots, x_n \mapsto l_n$ that maps output channels $x_i \in \mathrm{dom}\,(\Gamma_{\mathrm{out}})$ to heap locations. That is, N records for each output channel the location of the delayed computation that will produce the next value of the output channel as soon as it needs updating.

The machine can make three kinds of transitions:

an initialisation transition	$\langle t; \iota \rangle \overset{O}{\Longrightarrow} \langle N; \eta; \iota \rangle$
an input transition	$\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota' \rangle$
an output transition	$\langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle \overset{O}{\Longrightarrow} \langle N'; \eta_L; \iota \rangle$

where O is a sequence $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ that maps output channels to values. After the initial transition, which initialises the values of all output channels, the machine alternates between input transitions, each of which updates the value of an input channel and possibly the input buffer (if the new input is on a buffered channel), and output transitions, each of which provides new values for all those output channels triggered by the immediately preceding input transition.

The initialisation transition evaluates the reactive program t in the context of the initial input buffer ι and thereby produces a tuple $\langle v_1 :: l_1, \ldots, v_m :: l_m \rangle$ whose components $v_i :: l_i$ correspond to the output channels $x_i : A_i \in \Gamma_{\text{out}}$. Each v_i is the initial value of the output channel x_i and each l_i points to a delayed computation in the heap η that computes future values of x_i.

An input transition receives an updated value v on the input channel κ and reacts by updating the input buffer (if it already had a value for κ) and transitioning the store η to the new store $[\eta]_{\kappa\in} \langle \kappa \mapsto v \rangle [\eta]_{\kappa\notin}$. This splits the heap η into a part that contains κ and a part that does not:

$$[\eta]_{\kappa\in}(l) = \eta(l) \quad \text{if } \kappa \in \mathsf{cl}(l) \qquad\qquad [\eta]_{\kappa\notin}(l) = \eta(l) \quad \text{if } \kappa \notin \mathsf{cl}(l)$$

That is, in the subsequent output transition, the machine can read from $[\eta]_{\kappa\in}$, i.e., exactly those heap locations from η that were waiting for input from κ, and access the new value v from κ.

Finally, the output transition checks for each element $x \mapsto l$ in N, whether it should be advanced because it depends on κ (OUTPUT-COMPUTE) or should remain untouched because it does not depend on κ (OUTPUT-SKIP). Only in the OUTPUT-COMPUTE case, a new output value for x is produced. In the end, the output transition performs the desired garbage collection that deletes both the now heap η_N and the input value v (OUTPUT-END). This also means that the updates performed by OUTPUT-COMPUTE, are not only possible (because the required data arrived), but also necessary (because both the input data and the delayed computations they depend on will be gone after this output transition of the machine).

4.3 Example

To see the operational semantics in action, we revisit the simple GUI program from section 3.3 and run it on the machine. To this end, we first elaborate the definition of *toggleSig* into an explicit fixed point term of the core calculus as described in section 3.5:

$$\begin{aligned} \mathit{toggleSig} = \mathsf{fix}\, r.\lambda \mathit{tog}.\lambda f.\lambda g.\lambda x.& \\ \mathsf{let}\ \mathit{tick} = \mathsf{unbox}\ \mathit{tog}\ \mathsf{in}\ \mathit{switchf}\ (\mathsf{unbox}\ f\ x)&(\mathsf{delay}_{\mathsf{cl}(\mathit{tick})}\ (\mathit{adv}\ \mathit{tick}; \mathsf{adv}\ r\ \mathit{tog}\ g\ f)) \end{aligned}$$

During the execution, the machine turns fixed points like *toggleSig* into delayed fixed points that use dfix instead of fix. We write *toggleSig*′ for this delayed fixed point, i.e., *toggleSig*′ is obtained from *toggleSig* by replacing fix with dfix. We will use the same notational convention for other fixed point definitions and write $\mathit{sigAwait}'_\kappa$ and *scan*′ for the dfix versions of $\mathit{sigAwait}_\kappa$ and *scan* from section 3.1.

We consider the program $\mathit{field1} : \Delta \Rightarrow \Gamma_{\text{out}}$ with $\Delta = \{\mathsf{up} :_{\mathsf{p}} 1, \mathsf{toggle} :_{\mathsf{p}} 1\}$, $\Gamma_{\text{out}} = x : \mathsf{Nat}$, and

$$\begin{aligned} \mathit{field1} = \mathit{toggleSig}\ t\ s_1\ s_2\ 0 \quad \text{where} \quad t &= \mathsf{box}\, \mathsf{wait}_{\mathsf{toggle}} \\ s_1 &= \mathsf{box}\, (\mathit{count}\ \mathit{sigAwait}_{\mathsf{up}}) \\ s_2 &= \mathsf{box}\, \mathit{const} \end{aligned}$$

That is, this program describes the behaviour of the text field that initially is in focus and thus reacts to the 'up' button.

For better clarity of the transition steps of the machine, we write the machine's store as just the list of its heap locations, and write the contents of the locations along with their clocks separately underneath. The first step of the machine performs the initialisation that provides the initial value

of the output signal:

$$\langle \mathit{field1}; \emptyset \rangle \overset{x \mapsto 0}{\Longrightarrow} \langle x \mapsto l_1; l_1, l_2, l_3, l_4; \emptyset \rangle$$

$$\begin{array}{lll} \text{where} & l_1 \mapsto \mathsf{case}\ \mathsf{select}\ l_3\ l_4\ \mathsf{of}\ \ldots & \mathsf{cl}\,(l_1) = \{\mathsf{toggle}, \mathsf{up}\} \\ & l_2 \mapsto \mathsf{adv}\ \mathsf{wait}_{\mathsf{up}} :: \mathsf{adv}\ \mathit{sigAwait}' & \mathsf{cl}\,(l_2) = \{\mathsf{up}\} \\ & l_3 \mapsto \mathit{scan}\ (\mathsf{box}\ \lambda n.\lambda m.m + 1)\ 0\ (\mathsf{adv}\ l_2) & \mathsf{cl}\,(l_3) = \{\mathsf{up}\} \\ & l_4 \mapsto \mathsf{adv}\ \mathsf{wait}_{\mathsf{toggle}}; \mathsf{adv}\ \mathit{toggleSig}'\ t\ s_2\ s_1 & \mathsf{cl}\,(l_4) = \{\mathsf{toggle}\} \end{array}$$

We can see that the next value for the output channel x is provided by the delayed computation at location l_1, and since $\mathsf{cl}\,(l_1) = \{\mathsf{toggle}, \mathsf{up}\}$ we know that x will produce a new value as soon as the user clicks either of the two buttons. If the user clicks the 'up' button, we see the following:

$$\langle x \mapsto l_1; l_1, l_2, l_3, l_4; \emptyset \rangle \overset{\mathsf{up} \mapsto \langle\rangle}{\Longrightarrow} \langle x \mapsto l_1; l_1, l_2, l_3\ \langle \mathsf{up} \mapsto \langle\rangle \rangle\ l_4; \emptyset \rangle \overset{x \mapsto 1}{\Longrightarrow} \langle x \mapsto l_5; l_4, l_5, l_6, l_7; \emptyset \rangle$$

$$\begin{array}{lll} \text{where} & l_5 \mapsto \mathsf{case}\ \mathsf{select}\ l_7\ l_4\ \mathsf{of}\ \ldots & \mathsf{cl}\,(l_5) = \{\mathsf{toggle}, \mathsf{up}\} \\ & l_6 \mapsto \mathsf{adv}\ \mathsf{wait}_{\mathsf{up}} :: \mathsf{adv}\ \mathit{sigAwait}' & \mathsf{cl}\,(l_6) = \{\mathsf{up}\} \\ & l_7 \mapsto \mathsf{adv}\ \mathit{scan}'\ (\mathsf{box}\ \lambda n.\lambda m.m + 1)\ 0\ (\mathsf{adv}\ l_6) & \mathsf{cl}\,(l_7) = \{\mathsf{up}\} \end{array}$$

The heap locations l_1, l_2, l_3 are garbage collected and only l_4 survives since only the clock of l_4 does not contain up. If the user now clicks the 'toggle' button, we see the following:

$$\langle x \mapsto l_5; l_4, l_5, l_6, l_7; \emptyset \rangle \overset{\mathsf{toggle} \mapsto \langle\rangle}{\Longrightarrow} \langle x \mapsto l_5; l_4, l_5\ \langle \mathsf{toggle} \mapsto \langle\rangle \rangle\ l_6, l_7; \emptyset \rangle \overset{x \mapsto 1}{\Longrightarrow} \langle x \mapsto l_8; l_6, l_7, l_8, l_9; \emptyset \rangle$$

$$\begin{array}{lll} \text{where} & \mathsf{cl}\,(l_0) = \emptyset & l_8 \mapsto \mathsf{case}\ \mathsf{select}\ l_0\ l_9\ \mathsf{of}\ \ldots & \mathsf{cl}\,(l_8) = \{\mathsf{toggle}\} \\ & & l_9 \mapsto \mathsf{adv}\ \mathsf{wait}_{\mathsf{toggle}} :: \mathsf{adv}\ \mathit{sigAwait}'\ t\ s_1\ s_2 & \mathsf{cl}\,(l_9) = \{\mathsf{toggle}\} \end{array}$$

The heap location l_0 is allocated by never and thus does not appear on the heap. Now the output channel x only depends on the input channel toggle. If the user now repeatedly clicks the 'up' button, no output is produced:

$$\begin{aligned} \langle x \mapsto l_8; l_6, l_7, l_8, l_9; \emptyset \rangle &\overset{\mathsf{up} \mapsto \langle\rangle}{\Longrightarrow} \langle x \mapsto l_8; l_6, l_7\ \langle \mathsf{up} \mapsto \langle\rangle \rangle\ l_8, l_9; \emptyset \rangle \overset{\cdot}{\Longrightarrow} \langle x \mapsto l_8; l_8, l_9; \emptyset \rangle \\ &\overset{\mathsf{up} \mapsto \langle\rangle}{\Longrightarrow} \langle x \mapsto l_8; \langle \mathsf{up} \mapsto \langle\rangle \rangle\ l_8, l_9; \emptyset \rangle \overset{\cdot}{\Longrightarrow} \langle x \mapsto l_8; l_8, l_9; \emptyset \rangle \end{aligned}$$

Finally, note that since the input context Δ contains no buffered input channels the input buffer remains empty during the entire run of the program.

4.4 Main Results

The operational semantics presented above allows us to precisely state the operational guarantees provided by Async RaTT, namely productivity, the absence of implicit space leaks, causality, and signal independence. We address each of them in turn.

4.4.1 Productivity. Reactive programs $t : \Delta \Rightarrow \Gamma_{\mathsf{out}}$ are productive in the sense that if we feed t with a well-typed initial input buffer and an infinite sequence of well-typed inputs on its input channels, then it will produce an infinite sequence of well-typed outputs on its output channels. Before we can state the productivity property formally, we need to make precise what we mean by well-typed:

- An input buffer ι is well-typed , denoted $\vdash \iota : \Delta$, if $\vdash \iota(\kappa) : A$ for each κ such that $\kappa :_{\mathsf{b}} A \in \Delta$ or $\kappa :_{\mathsf{bp}} A \in \Delta$.
- An input value $\kappa \mapsto v$ is well-typed, written $\vdash \kappa \mapsto v : \Delta$, if $\kappa :_c A \in \Delta$ and $\vdash v : A$.

- A set of output values O is well-typed, written $\vdash O : \Gamma_{\text{out}}$, if for all $x \mapsto v \in O$, we have that $x : A \in \Gamma_{\text{out}}$ and $\vdash v : A$.

We can now formally state the productivity property as follows:

Theorem 4.1 (productivity)**.** Given a reactive program $t : \Delta \Rightarrow \Gamma_{\text{out}}$, well-typed input values $\vdash \kappa_i \mapsto v_i : \Delta$ for all $i \in \mathbb{N}$, and a well-typed initial input buffer $\vdash \iota_0 : \Delta$, there is an infinite transition sequence

$$\langle t; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle N_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle N_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle N_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots$$

with $\vdash O_i : \Gamma_{\text{out}}$ for all $i \in \mathbb{N}$.

While a reactive program will always produce a set of output values O_{i+1} for each incoming input value $\kappa_i \mapsto v_i$, this set may be empty. This happens if none of the heap locations in N_i depends on the input κ_i, i.e., if $\kappa_i \notin \mathsf{cl}(l)$ for all $x \mapsto l \in N_i$. As we will see in Proposition 4.4, this will necessarily be the case for inputs $\kappa :_{\mathsf{b}} A \in \Delta$ that are buffered-only. Note that all output channels are initialised in the initialisation transition. An empty set of output values therefore only means that no output channels need to be updated.

4.4.2 Implicit Space Leaks. The absence of implicit space leaks is a direct consequence of the productivity property (Theorem 4.1). More precisely, the operational semantics of Async RaTT is formulated in such a way that after each pair of input/output transitions

$$\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota' \rangle \overset{O}{\Longrightarrow} \langle N'; \eta_L; \iota' \rangle$$

all heap locations l in η that depend on κ, i.e., those with $\kappa \in \mathsf{cl}(l)$, are garbage collected and thus do not appear in η_L. That is, a delayed computation at location l is only kept in memory until its clock $\mathsf{cl}(l)$ ticks. By Theorem 4.1, this aggressive garbage collection strategy is safe: The machine never gets stuck attempting to dereference a garbage collected heap location.

4.4.3 Causality. In the following we refer to the transition sequences for a reactive program t obtained by Theorem 4.1 simply as well-typed transition sequences for t.

A reactive program t is causal, if for any of its well-typed transition sequences

$$\langle t; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle N_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle N_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle N_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots \tag{1}$$

each set of output values O_n only depends on the initial input buffer ι_0 and previously received input values $\kappa_i \mapsto v_i$ with $i < n$. To see that this is always the case, we first note that the operational semantics is deterministic in the following sense:

Lemma 4.2 (deterministic semantics)**.**

(i) $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v_1; \sigma_1 \rangle$ *and* $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v_2; \sigma_2 \rangle$ *implies that* $v_1 = v_2$ *and that* $\sigma_1 = \sigma_2$.
(ii) $c \overset{\kappa \mapsto v}{\Longrightarrow} c_1$ *and* $c \overset{\kappa \mapsto v}{\Longrightarrow} c_2$ *implies* $c_1 = c_2$.
(iii) $c \overset{O_1}{\Longrightarrow} c_1$ *and* $c \overset{O_2}{\Longrightarrow} c_2$ *implies* $O_1 = O_2$ *and* $c_1 = c_2$.

Causality now follows from Theorem 4.1 and Lemma 4.2.

Corollary 4.3 (causality)**.** *Suppose (1) as well as the following are well-typed transition sequences*

$$\langle t; \iota_0 \rangle \overset{O'_0}{\Longrightarrow} \langle N'_0; \eta'_0; \iota'_0 \rangle \overset{\kappa'_0 \mapsto v'_0}{\Longrightarrow} \langle N'_0; \sigma'_0; \iota'_1 \rangle \overset{O'_1}{\Longrightarrow} \langle N'_1; \eta'_1; \iota'_1 \rangle \overset{\kappa'_1 \mapsto v'_1}{\Longrightarrow} \dots$$

Let $n \in \mathbb{N}$ *and suppose* $\kappa'_i = \kappa_i$ *and* $v'_i = v_i$ *for all* $i < n$. *Then* $O'_n = O_n$.

4.4.4 Signal Independence. From the definition of the reactive semantics we can see that the machine only updates an output channel $x : A \in \Gamma_{\text{out}}$ if it depends on the input value $\kappa \mapsto v$ that has just arrived, i.e., if the machine is in a state $\langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle$ with $\kappa \in \mathsf{cl}(N(x))$. However, the typing system allows us to give two useful *static* criteria for when $\kappa \notin \mathsf{cl}(N(x))$ is guaranteed and thus the output signal x need not (and indeed cannot) be updated.

As alluded to earlier, values received on buffered-only channels will never produce an output:

Proposition 4.4 (buffered signal independence)**.** *Suppose $t : \Delta \Rightarrow \Gamma_{\text{out}}$ is a reactive program and*

$$\langle t; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle N_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle N_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle N_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots$$

is a well-typed transition sequence for t. Then O_{i+1} is empty whenever $\kappa_i :_{\mathrm{b}} A \in \Delta$ for some A.

Secondly, the input context Δ for a given output signal implementation gives us an upper bound on the push channels that will trigger an update:

Theorem 4.5 (push signal independence)**.** Suppose $(t, s) : \Delta \Rightarrow \Gamma_{\text{out}}$ is a reactive program with $\Gamma_{\text{out}} = \Gamma, z : C$ such that also $s : \Delta' \Rightarrow (z : C)$ is a reactive program for some $\Delta' \subset \Delta$ and

$$\langle (t, s)\,; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle N_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle N_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle N_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots$$

is a well-typed transition sequence for (t, s). Then $z \mapsto v \in O_{i+1}$ implies that $\kappa_i \in \mathsf{dom}(\Delta')$. In other words, the output channel z is only updated when inputs in Δ' are updated.

5 METATHEORY

In this section, we sketch the proof of the operational properties presented in section 4.4, namely Theorem 4.1, Proposition 4.4, and Theorem 4.5. All three follow from a more general semantic soundness property. To prove this property, we first devise a semantic model of the Async RaTT calculus in the form of a Kripke logical relation. That is, the model consists of a family $[\![A]\!](w)$ of sets of closed terms that satisfy the soundness properties we are interested in. This family of sets is indexed by a *world* and is defined by induction on the structure of the type A and world w. The soundness proof is thus reduced to a proof that $\vdash_\Delta t : A$ implies $t \in [\![A]\!](w)$, which is also known as the *fundamental property* of the logical relation.

5.1 Kripke Logical Relation

The worlds w for our logical relation consist of two components: a natural number n and a store σ. The number n allows us to model guarded recursive types via step-indexing [Appel and McAllester 2001]. This is achieved by defining $[\![\triangleleft A]\!](n+1, \sigma)$ in terms of $[\![A]\!](n, \sigma')$ for some suitable σ'. Since recursive types $\mathsf{Fix}\,\alpha.A$ unfold to $A[\triangleleft(\mathsf{Fix}\,\alpha.A)/\alpha]$, we can define $[\![\mathsf{Fix}\,\alpha.A]\!](n+1, \sigma)$ in terms of $[\![A]\!](n+1, \sigma)$ and $[\![\mathsf{Fix}\,\alpha.A]\!](n, \sigma')$, which is well-founded since in the former we refer to the smaller type A and in the latter we refer to a smaller step index n.

A key aspect of the operational semantics of Async RaTT is that it stores delayed computations in a store σ. Hence, in order to capture the semantics of a term t, we have to account for the fact that t may contain heap locations that point into some suitable store σ. Intuitively speaking, the set $[\![A]\!](n, \sigma)$ contains those terms that, starting with the store σ, can be executed safely to produce a value of type A. Ultimately, the index σ enables us to prove that the garbage collection performed by the reactive semantics is indeed sound.

What makes $[\![A]\!](n, \sigma)$ a Kripke logical relation is the fact that we have a preorder $\lesssim$ on worlds such that $(n, \sigma) \lesssim (n', \sigma')$ implies $[\![A]\!](n, \sigma) \subseteq [\![A]\!](n', \sigma')$. We can think of (n', σ') as a future world reachable from (n, σ), i.e., it describes how the surrounding context changes as the machine performs computations. There are four different kinds of changes, which we address in turn below:

Firstly, time may pass, which means that we have fewer time steps left, i.e., $n > n'$. Secondly, the machine performs garbage collection on the store σ. The following garbage collection function describes this:

$$\mathrm{gc}(\eta_L) = \eta_L \qquad \mathrm{gc}(\eta_N \langle \kappa \mapsto v \rangle \eta_L) = \eta_L$$

Third, the machine may store delayed computation in σ, which we account for by the order $\sqsubseteq$ on heaps and stores:

$$\frac{\eta(l) = \eta'(l) \text{ for all } l \in \mathrm{dom}\,(\eta)}{\eta \sqsubseteq \eta'} \qquad \frac{\eta_N \sqsubseteq \eta'_N \qquad \eta_L \sqsubseteq \eta'_L}{\eta_N \langle \kappa \mapsto v \rangle \eta_L \sqsubseteq \eta'_N \langle \kappa \mapsto v \rangle \eta'_L}$$

That is, $\sigma \sqsubseteq \sigma'$ iff σ' is obtained from σ by storing additional terms.

Finally, the machine may receive an input value $\kappa \mapsto v$, which is captured by the following order $\sqsubseteq^{\Delta}_{\checkmark}$ on stores:

$$\frac{\sigma \sqsubseteq \sigma'}{\sigma \sqsubseteq^{\Delta}_{\checkmark} \sigma'} \qquad \frac{\eta_L \sqsubseteq \eta'_L \qquad \kappa :_c A \in \Delta \qquad \vdash v : A}{\eta_L \sqsubseteq^{\Delta}_{\checkmark} \eta_N \langle \kappa \mapsto v \rangle \eta'_L}$$

That is, in addition to the allocations captured by $\sqsubseteq$, the order may also introduce an input value $\kappa \mapsto v$.

Taken together, we can define the Kripke preorder $\lesssim$ on worlds as follows:

$$(n, \sigma) \lesssim (n', \sigma') \quad \text{iff} \quad n \geq n' \text{ and } \sigma \sqsubseteq^{\Delta}_{\checkmark} \sigma'$$

This does not include garbage collection, as it is restricted to certain circumstances. Indeed, the machine performs garbage collection only at certain points of the execution, namely at the end of an output transition.

Finally, before we can give the definition of the Kripke logical relation, we need to semantically capture the notion of input independence that is needed both for the operational semantics of select and the signal independence properties (Proposition 4.4 and Theorem 4.5). In essence, we need that a heap location l in the world $(n+1, \sigma)$ should still be present in the future world (n, σ') in which we received an input on a channel $\kappa \notin l$. We achieve this by making the logical relation $[\![\circledcirc A]\!](n, \sigma)$ satisfy the following clock independence property:

$$\text{If } l \in [\![\circledcirc A]\!](n, \sigma), \text{ then } l \in [\![\circledcirc A]\!](n, [\sigma]_{\mathrm{cl}(l)})$$

where $[\sigma]_\Theta$ restricts σ to heap locations whose clocks are *subclocks* of Θ:

$$[\eta]_\Theta(l) = \eta(l) \qquad \text{if } \mathrm{cl}\,(l) \subseteq \Theta$$

$$[\eta_N \langle \kappa \mapsto v \rangle \eta_L]_\Theta = \begin{cases} [\eta_N]_\Theta \langle \kappa \mapsto v \rangle [\eta_L]_\Theta & \text{if } \kappa \in \Theta \\ [\eta_L]_\Theta & \text{if } \kappa \notin \Theta \end{cases}$$

The full definition of the Kripke logical relation is given in Figure 7. In addition to the aspects discussed above, it is parameterised by the context Δ and distinguishes between the value relation $\mathcal{V}_\Delta[\![A]\!](w)$ and the term relation $\mathcal{T}_\Delta[\![A]\!](w)$. The two relations are defined by well-founded recursion by the lexicographic ordering on the tuple $(n, |A|, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$|\alpha| = |\circledcirc A| = |\circledv A| = |1| = |\mathsf{Nat}| = 1$$
$$|A \times B| = |A + B| = |A \to B| = 1 + |A| + |B|$$
$$|\square A| = |\mathsf{Fix}\,\alpha.A| = 1 + |A|$$

Note that in the definition for $\mathcal{V}_\Delta[\![\circledcirc A]\!](w)$ in Figure 7, we use the shorthand $\sigma(l)$ for $\eta_L(l)$, where η_L is the later heap of σ.

$$
\begin{aligned}
\mathcal{V}_\Delta[\![1]\!](w) &= \{\langle\rangle\}, \\
\mathcal{V}_\Delta[\![\mathsf{Nat}]\!](w) &= \left\{\mathsf{suc}^n\, 0 \,\middle|\, n \in \mathbb{N}\right\}, \\
\mathcal{V}_\Delta[\![A \times B]\!](w) &= \left\{(v_1, v_2) \,\middle|\, v_1 \in \mathcal{V}_\Delta[\![A]\!](w) \wedge v_2 \in \mathcal{V}_\Delta[\![B]\!](w)\right\}, \\
\mathcal{V}_\Delta[\![A + B]\!](w) &= \left\{\mathsf{in}_1\, v \,\middle|\, v \in \mathcal{V}_\Delta[\![A]\!](w)\right\} \cup \left\{\mathsf{in}_2\, v \,\middle|\, v \in \mathcal{V}_\Delta[\![B]\!](w)\right\} \\
\mathcal{V}_\Delta[\![A \to B]\!](n, \sigma) &= \left\{\lambda x.t \,\middle|\, \forall \sigma' \sqsupseteq^\Delta_\checkmark \mathrm{gc}(\sigma), n' \le n, v \in \mathcal{V}_\Delta[\![A]\!](n', \sigma').t[v/x] \in \mathcal{T}_\Delta[\![B]\!](n', \sigma')\right\} \\
\mathcal{V}_\Delta[\![\Box A]\!](n, \sigma) &= \left\{\mathsf{box}\, t \,\middle|\, t \in \mathcal{T}_\Delta[\![A]\!](n, \emptyset)\right\} \\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize V} A]\!](0, \sigma) &= \{\mathsf{dfix}\, x.t \mid \mathsf{dfix}\, x.t \text{ a closed term}\} \\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize V} A]\!](n+1, \sigma) &= \left\{\mathsf{dfix}\, x.t \,\middle|\, t[\mathsf{dfix}\, x.t/x] \in \mathcal{T}_\Delta[\![A]\!](n, \emptyset)\right\} \\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize $\exists$} A]\!](0, \sigma) &= \mathsf{Loc}_\Delta \cup \{\mathsf{wait}_\kappa \mid \kappa :_c A \in \Delta, c \in \{\mathsf{p}, \mathsf{bp}\}\} \\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize $\exists$} A]\!](n+1, \sigma) &= \left\{l \in \mathsf{Loc}_\Delta \,\middle|\, \forall \kappa \in \mathrm{cl}\,(l), \vdash v : \Delta(\kappa).\sigma(l) \in \mathcal{T}_\Delta[\![A]\!](n, \left[[\mathrm{gc}(\sigma)]_{\kappa \in} \langle \kappa \mapsto v \rangle [\mathrm{gc}(\sigma)]_{\kappa \notin}\right]_{\mathrm{cl}(l)})\right\} \\
&\quad \cup \{\mathsf{wait}_\kappa \mid \kappa :_c A \in \Delta, c \in \{\mathsf{p}, \mathsf{bp}\}\} \\
\mathcal{V}_\Delta[\![\mathsf{Fix}\, \alpha.A]\!](w) &= \left\{\mathsf{into}\, v \,\middle|\, v \in \mathcal{V}_\Delta[\![A[\textcircled{\scriptsize $\exists$}(\mathsf{Fix}\, \alpha.A)/\alpha]]\!](w)\right\}
\end{aligned}
$$

$$
\mathcal{T}_\Delta[\![A]\!](n, \sigma) = \left\{t \,\middle|\, t \text{ closed}, \forall \iota : \Delta. \forall \sigma' \sqsupseteq^\Delta_\checkmark \sigma. \exists \sigma'', v.\, \langle t; \sigma' \rangle \Downarrow^\iota \langle v; \sigma'' \rangle \wedge v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')\right\}
$$

$$
\begin{aligned}
C_\Delta[\![\cdot]\!](w) &= \{*\} \\
C_\Delta[\![\Gamma, x : A]\!](w) &= \left\{\gamma[x \mapsto v] \,\middle|\, \gamma \in C_\Delta[\![\Gamma]\!](w), v \in \mathcal{V}_\Delta[\![A]\!](w)\right\} \\
C_\Delta[\![\Gamma, \checkmark_\theta]\!](n, \eta_N \langle \kappa \mapsto v \rangle \eta_L) &= \left\{\gamma \in C_\Delta[\![\Gamma]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin})) \,\middle|\, \begin{array}{l} [\eta_L]_\Theta = \left[\eta'_L\right]_\Theta, [\eta_N]_\Theta = \left[\eta'_N\right]_\Theta, \\ \eta'_N \in \mathsf{Heap}^\kappa, \vdash v : \Delta(\kappa), \kappa \in \Theta, \text{ and} \\ \Theta \subseteq \mathrm{dom_p}\,(\Delta), \text{ where } \Theta = |\theta\gamma| \end{array}\right\}
\end{aligned}
$$

Using

$$
\mathrm{dom_p}\,(\Delta) = \{\kappa \mid \exists A, c \in \{\mathsf{p}, \mathsf{bp}\}. x :_c A \in \Delta\} \qquad \mathsf{Loc}_\Delta = \left\{l \in \mathsf{Loc} \,\middle|\, \mathrm{cl}\,(l) \subseteq \mathrm{dom_p}\,(\Delta)\right\}
$$

Fig. 7. Logical relation.

Our goal is to prove the fundamental property, i.e., that $\vdash_\Delta t : A$ implies $t \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, by induction on the typing derivation. Therefore, we need to generalise the fundamental property to open terms as well. That means we also need a corresponding logical relation for contexts, which is given at the bottom of Figure 7. The interpretation of $\checkmark_\theta$ in a context is quite technical, but is essentially determined by the interpretation of $\textcircled{\scriptsize $\exists$}$ due to the requirement of being left adjoint [Birkedal et al. 2020].

The three logical relations indeed satisfy the preservation under the Kripke preorder $\lesssim$.

Lemma 5.1. *Let $n \ge n'$ and $\sigma \sqsubseteq^\Delta_\checkmark \sigma'$.*

(i) $\mathcal{V}_\Delta[\![A]\!](n, \sigma) \subseteq \mathcal{V}_\Delta[\![A]\!](n', \sigma')$.
(ii) $\mathcal{T}_\Delta[\![A]\!](n, \sigma) \subseteq \mathcal{T}_\Delta[\![A]\!](n', \sigma')$.
(iii) $C_\Delta[\![\Gamma]\!](n, \sigma) \subseteq C_\Delta[\![\Gamma]\!](n', \sigma')$.

Preservation under garbage collection, however, only holds for values and tick-free contexts:

Lemma 5.2. *(garbage collection)*

(i) $\mathcal{V}_\Delta[\![A]\!](n, \sigma) \subseteq \mathcal{V}_\Delta[\![A]\!](n, \mathrm{gc}(\sigma))$.
(ii) $C_\Delta[\![\Gamma]\!](n, \sigma) \subseteq C_\Delta[\![\Gamma]\!](n, \mathrm{gc}(\sigma))$ *if* Γ *is tick-free.*

Moreover, the clock independence property holds for both the value and context relations:

Lemma 5.3.

(i) If $v \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n,\sigma)$*, then* $v \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n,\sigma')$*, for any* σ' *with* $[\sigma]_{\mathrm{cl}(v)} = [\sigma']_{\mathrm{cl}(v)}$.
(ii) If $\gamma \in C_\Delta[\![\Gamma, \checkmark_\theta]\!](n,\sigma)$*, then* $\gamma \in C_\Delta[\![\Gamma, \checkmark_\theta]\!](n, [\sigma]_{|\theta\gamma|})$

Finally, we obtain the soundness of the language by the following fundamental property of the logical relation:

Theorem 5.4. Given $\Gamma \vdash_\Delta$, $\Gamma \vdash_\Delta t : A$, and $\gamma \in C_\Delta[\![\Gamma]\!](n,\sigma)$, then $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$.

The proof is a standard induction on the typing relation $\Gamma \vdash_\Delta t : A$ that makes use of the aforementioned closure properties of the logical relations and is included in Appendix A.

5.2 Operational Properties

We close this section by showing how we can use the fundamental property to prove the operational properties presented in section 4.4. To this end, we will sketch the proofs of Theorem 4.1, Proposition 4.4, and Theorem 4.5.

5.2.1 Productivity. In the following we assume a fixed reactive interface $\Delta \Rightarrow \Gamma_{\mathrm{out}}$, for which we define the following sets of machine states I_n and S_n for the reactive semantics:

$$I_n = \left\{ \langle t; \iota \rangle \,\middle|\, \iota : \Delta \wedge t \in \mathcal{T}_\Delta[\![\mathrm{Prod}\,(\Gamma_{\mathrm{out}})]\!](n,\emptyset) \right\}$$
$$S_n = \left\{ \langle N; \eta; \iota \rangle \,\middle|\, \iota : \Delta \wedge \forall x \mapsto l \in N. \exists x : A \in \Gamma_{\mathrm{out}}. l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\,A)]\!](n,\eta) \right\}$$

The following lemma proves that the machine stays inside the sets of states defined above and will only produce well-typed outputs. For the latter, we make use of the fact that $v \in \mathcal{V}_\Delta[\![A]\!](n,\sigma)$ iff $\vdash v : A$ for every value type A.

Lemma 5.5 (productivity)**.**

(i) If $t : \Delta \Rightarrow \Gamma_{\mathrm{out}}$ *and* $\iota : \Delta$*, then* $\langle t; \iota \rangle \in I_n$ *for all* $n \in \mathbb{N}$.
(ii) If $\langle t; \iota \rangle \in I_n$*, then there is a transition* $\langle t; \iota \rangle \overset{O}{\Longrightarrow} \langle N; \eta; \iota \rangle$ *such that* $\langle N; \eta; \iota \rangle \in S_n$ *and* $\vdash O : \Gamma_{\mathrm{out}}$.
(iii) If $\langle N; \eta; \iota \rangle \in S_{n+1}$ *and* $\vdash \kappa \mapsto v : \Delta$*, then there is a sequence of two transitions*

$$\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle N; \sigma; \iota' \rangle \overset{O}{\Longrightarrow} \langle N'; \eta'; \iota' \rangle$$

such that $\langle N'; \eta'; \iota' \rangle \in S_n$ *and* $\vdash O : \Gamma_{\mathrm{out}}$.

Proof.

(i) We need to show that $t \in \mathcal{T}_\Delta[\![\mathrm{Prod}\,(\Gamma_{\mathrm{out}})]\!](n,\emptyset)$. Since $t : \Delta \Rightarrow \Gamma_{\mathrm{out}}$, we know that $\vdash_\Delta t : \mathrm{Prod}\,(\Gamma_{\mathrm{out}})$. Hence, by Theorem 5.4, we have that $t \in \mathcal{T}_\Delta[\![\mathrm{Prod}\,(\Gamma_{\mathrm{out}})]\!](n,\emptyset)$.

(ii) Let $\langle t; \iota \rangle \in I_n$. We thus have $t \in \mathcal{T}_\Delta[\![\mathrm{Prod}\,(\Gamma_{\mathrm{out}})]\!](n,\emptyset)$. Therefore, by definition, we have $\langle t; \emptyset \rangle \Downarrow^\iota \langle \langle v_1 :: w_1, \ldots, v_k :: w_k \rangle ; \eta \rangle$ with $v_i \in \mathcal{V}_\Delta[\![A_i]\!](n,\eta)$ and $w_i \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\,A_i)]\!](n,\eta)$. Since $\mathrm{Sig}\,A_i$ is not a value type, we know that each w_i must be a heap location, so we can write l_i for w_i. Hence, by definition, $\langle t; \iota \rangle \overset{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k}{\Longrightarrow} \langle x_1 \mapsto l_1, \ldots, x_k \mapsto l_k; \eta; \iota \rangle$ and $\langle x_1 \mapsto l_1, \ldots, x_k \mapsto l_k; \eta; \iota \rangle \in S_n$. Since each A_i is a value type, we know that $v_i \in \mathcal{V}_\Delta[\![A_i]\!](n,\eta)$ implies $\vdash v_i : A_i$ and thus $\vdash x_1 \mapsto v_1, \ldots, x_k \mapsto v_k : \Gamma_{\mathrm{out}}$.

(iii) By definition $\langle N;\eta;\iota\rangle \overset{\kappa\mapsto v}{\Longrightarrow} \langle N;\sigma;\iota'\rangle$, where $\sigma = [\eta]_{\kappa\in}\,\langle\kappa\mapsto v\rangle\,[\eta]_{\kappa\notin}$. We claim the following three statements:

$$\text{If } y : B \in \Gamma_{\text{out}}, N(y) = l, \kappa \in \mathsf{cl}\,(l)\,,\ \text{then } \mathsf{adv}\, l \in \mathcal{T}_\Delta[\![\mathrm{Sig}\, B]\!](n,\sigma) \tag{2}$$

$$\text{If } y : B \in \Gamma_{\text{out}}, N(y) = l, \kappa \notin \mathsf{cl}\,(l)\,,\ \text{then } l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, B)]\!](n,\sigma) \tag{3}$$

$$\text{If } \sigma' \sqsupseteq \sigma, N_1 \subseteq N,\ \text{then } \langle N_1;\sigma';\iota'\rangle \overset{O}{\Longrightarrow} \langle N_2;\eta';\iota'\rangle, \langle N_2;\eta';\iota'\rangle \in S_n, \mathrm{gc}(\sigma') \sqsubseteq \eta', \vdash O : \Gamma_{\text{out}} \tag{4}$$

From (4), we then obtain that $\langle N;\sigma;\iota'\rangle \overset{O}{\Longrightarrow} \langle N';\eta';\iota'\rangle$, with $\langle N';\eta';\iota'\rangle \in S_n$ and $\vdash O : \Gamma_{\text{out}}$. We conclude this proof by proving the above three claims:

- Proof of (2). Since $\langle N;\eta;\iota\rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, B)]\!](n+1,\eta)$ and thus $\mathsf{adv}\, l \in \mathcal{T}_\Delta[\![\mathrm{Sig}\, B]\!](n,\sigma)$.
- Proof of (3). Since $\langle N;\eta;\iota\rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, B)]\!](n+1,\eta)$. Since $\kappa \notin \mathsf{cl}\,(l)$, we can conclude that $[\eta]_{\mathsf{cl}(l)} = \left[[\eta]_{\kappa\notin}\right]_{\mathsf{cl}(l)}$. In turn, by Lemma 5.3, this means that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, B)]\!](n+1,[\eta]_{\kappa\notin})$, which by Lemma 5.1 implies that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, B)]\!](n,\sigma)$.
- Proof of (4). We proceed by induction on N_1. The case $N_1 = \cdot$ is trivial. For the case $N_1 = y \mapsto w, N_1'$, we distinguish between the case where $\kappa \in \mathsf{cl}\,(w)$ and the case where $\kappa \notin \mathsf{cl}\,(w)$. Then we can apply (2) or (3), respectively, and use the induction hypothesis for N_1' and some $\sigma'' \sqsupseteq \sigma'$. If $\kappa \notin \mathsf{cl}\,(w)$, then σ'' is just σ'. If $\kappa \in \mathsf{cl}\,(w)$, then σ'' arises from the evaluation $\langle \mathsf{adv}\, w;\sigma'\rangle \Downarrow^{\iota'} \langle v :: w';\sigma''\rangle$ we obtained from (2). In either case, we make use of Lemma 5.1 and Lemma 5.2 to conclude $\langle N_2;\eta';\iota'\rangle \in S_n$. □

The productivity property is now a straightforward consequence of the above lemma:

Proof of Theorem 4.1 (productivity). For each $n \in \mathbb{N}$ we can we can construct the following finite transition sequence s_n using Lemma 5.5:

$$\langle t;\iota_0\rangle \overset{O_0}{\Longrightarrow} \langle N_0;\eta_0;\iota_0\rangle \overset{\kappa_0\mapsto v_0}{\Longrightarrow} \langle N_0;\sigma_0;\iota_1\rangle \overset{O_1}{\Longrightarrow} \langle N_1;\eta_1;\iota_1\rangle \overset{\kappa_1\mapsto v_1}{\Longrightarrow} \ldots \overset{O_n}{\Longrightarrow} \langle N_n;\eta_n;\iota_n\rangle$$

with $\vdash O_i : \Gamma_{\text{out}}$ for all $0 \le i \le n$. By Lemma 4.2, s_n is a prefix of s_m for all $m > n$. We thus obtain the desired infinite transition sequence as the limit of all s_n. □

5.2.2 Signal Independence. Proposition 4.4 is a straightforward consequence of Lemma 5.5:

Proof of Proposition 4.4 (buffered signal independence). By Lemma 5.5, for any pair of transitions $\langle N_i;\eta_i;\iota'\rangle \overset{\kappa_i\mapsto v_i}{\Longrightarrow} \langle N_i;\sigma_i;\iota_{i+1}\rangle \overset{O_{i+1}}{\Longrightarrow} \langle N_{i+1};\eta_{i+1};\iota_{i+1}\rangle$ in a sequence starting from $\langle t;\iota_i\rangle$, we have that $\langle N_i;\eta_i;\iota'\rangle \in S_1$. In particular, this means that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathrm{Sig}\, A)]\!](1,\eta_i)$ for any $x \mapsto l \in N_i$ with $x : A \in \Gamma_{\text{out}}$, which in turn means that $l \in \mathrm{Loc}_\Delta$. Hence, $\kappa_i \notin \mathsf{cl}\,(l)$, and so O_{i+1} is empty. □

For the proof of Theorem 4.5, we first define the following sets of machine states in the context of a partial map Δ' that maps variables x to input contexts Δ'_x:

$$T_n^{\Delta'} = \left\{ \langle N;\eta;\iota\rangle \,\middle|\, \iota : \Delta \wedge \forall x \mapsto w \in N.\Delta'_x \subseteq \Delta \wedge \exists x : A \in \Gamma_{\text{out}}.w \in \mathcal{V}_{\Delta'_x}[\![\textcircled{\exists}(\mathrm{Sig}\, A)]\!](n,\eta) \right\}$$

Machine states in $T_n^{\Delta'}$ are maintained during the execution of the machine:

Lemma 5.6. *If* $\langle N;\eta;\iota\rangle \in T_{n+1}^{\Delta'}$ *and* $\langle N;\eta;\iota\rangle \overset{\kappa\mapsto v}{\Longrightarrow} \langle N;\sigma;\iota'\rangle \overset{O}{\Longrightarrow} \langle N';\eta';\iota'\rangle$, *then* $\langle N';\eta';\iota'\rangle \in T_n^{\Delta'}$.

Proof. Note that $\sigma = [\eta]_{\kappa\in}\,\langle\kappa\mapsto v\rangle\,[\eta]_{\kappa\notin}$ and that $[\eta]_{\kappa\notin} \subseteq \eta'$. Suppose $(x \mapsto w) \in N$, and note that w must be a location, since $\mathrm{Sig}\, A$ is not a value type. We will write l for w to emphasise

this. Then there is an l' such that $(x \mapsto l') \in N'$ and we must show that $l' \in \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, \eta')$ using the hypothesis $l \in \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n+1, \eta)$. Suppose first that $\kappa \notin \mathrm{cl}\,(l)$. Then $l' = l$ and thus

$$l' \in \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, [\eta]_{\mathrm{cl}(l)}) \subseteq \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, [\eta]_{\kappa\notin}) \subseteq \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, \eta')$$

Suppose now $\kappa \in \mathrm{cl}\,(l)$. In that case, l' must have occurred by evaluating $\langle \sigma'(l); \sigma' \rangle \Downarrow^\iota \langle w :: l'; \sigma'' \rangle$ for some σ', σ'' such that $\sigma \subseteq \sigma'$, and $\mathrm{gc}(\sigma'') \subseteq \eta'$. The hypothesis tells us that

$$\eta(l) \in \mathcal{T}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, [\eta]_{\kappa\in}\, \langle \kappa \mapsto v \rangle\, [\eta]_{\kappa\notin})$$

Since $[\eta]_{\kappa\in}\, \langle \kappa \mapsto v \rangle\, [\eta]_{\kappa\notin} \subseteq \sigma'$, this means that $\langle \eta(l); \sigma' \rangle \Downarrow^\iota \langle w' :: l''; \sigma''' \rangle$. Since $\eta(l) = [\eta]_{\kappa\in}(l) = \sigma'(l)$, by the determinism of the operational semantics (Lemma 4.2), $\sigma''' = \sigma''$ and $l'' = l'$. From this we conclude that

$$l' \in \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, \sigma'') = \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, \mathrm{gc}(\sigma'')) \subseteq \mathcal{V}_{\Delta'_x}[\![\circledcirc(\mathrm{Sig}\, A)]\!](n, \eta') \qquad \square$$

Proof of Theorem 4.5 (push signal independence). Let $\Gamma_{\mathrm{out}} = x_1 : B_1, \ldots, x_m : B_m, z : C$. The initialisation transition

$$\langle (t, s)\,; \iota_0 \rangle \overset{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m, z \mapsto w}{\Longrightarrow} \langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m, z \mapsto l; \eta_0; \iota_0 \rangle = \langle N_0; \eta_0; \iota_0 \rangle$$

is caused by evaluations of the form $\langle t; \emptyset \rangle \Downarrow^\iota \langle \langle v_1 :: l_1, \ldots, v_m :: l_m \rangle\,; \eta \rangle$, and $\langle s; \eta \rangle \Downarrow^\iota \langle w :: l; \eta_0 \rangle$. By assumption $\vdash_{\Delta'} s : \mathrm{Sig}\, C$ and thus $s \in \mathcal{T}_{\Delta'}[\![\mathrm{Sig}\, C]\!](n, \emptyset)$ for all n by Theorem 5.4, which in turn implies $l \in \mathcal{V}_{\Delta'}[\![\circledcirc \mathrm{Sig}\, C]\!](n, \eta_0)$ for all n. Likewise $l_j \in \mathcal{V}_{\Delta}[\![\circledcirc \mathrm{Sig}\, B_j]\!](n, \eta_0)$ for all n, and $j = 1, \ldots, m$. So $\langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m, z \mapsto l; \eta_0; \iota_0 \rangle \in T_n^{\Delta''}$ for all n, where $\Delta''_z = \Delta'$ and $\Delta''_{x_j} = \Delta$, for $j = 1, \ldots, m$. By n applications of Lemma 5.6, we thus obtain that $\langle N_n; \eta_n; \iota_n \rangle \in T_1^{\Delta''}$. In particular, $N_n(z) = l'$ for some $l' \in \mathcal{V}_{\Delta'}[\![\circledcirc(\mathrm{Sig}\, C)]\!](1, \eta_n)$. Hence, $\mathrm{cl}\,(l') \subseteq \mathrm{dom}\,(\Delta')$ and thus $z \mapsto v \in O_{i+1}$ implies $\kappa_i \in \mathrm{cl}\,(l')$, which in turn implies $\kappa_i \in \mathrm{dom}\,(\Delta')$. $\square$

6 RELATED WORK

Functional reactive programming originates with Elliott and Hudak [1997]. The use of modal types for FRP was first suggested by Krishnaswami and Benton [2011], and the connection between linear temporal logic and FRP was discovered independently by Jeffrey [2012] and Jeltsch [2012]. Although some of these calculi have been implemented, they do not offer operational guarantees like the ones proved here for lack of space leaks. The first such operational guarantees were given by Krishnaswami et al. [2012] who describe a modal FRP language using linear types and allocation resources to statically bound the memory used by a reactive program. The simpler, but less precise, idea of using an aggressive garbage collection technique for avoiding space leaks is due to Krishnaswami [2013]. Krishnaswami's calculus used a dual context approach to programming with modal types. Bahr et al. [2019] recast these results in a Fitch-style modal calculus, the first in the RaTT family. This was later implemented in Haskell with some minor modifications [Bahr 2022].

All the above calculi are based on a global notion of time, which in almost all cases is discrete. In particular, the modal operator $\bigcirc$ for time steps in these calculi refers to the next time step on the global clock. One can of course also understand the step semantics of Async RaTT as operating on a global clock, but in our model each step is associated with an input coming from an input channel, and this allows us to define the delay modality $\circledcirc$ as a delay on a set of input channels. From the model perspective, $\circledcirc A$ carries some similarities with the type $\bigcirc(\diamond A)$, where $\diamond A \cong A + \bigcirc\diamond A$ is a guarded recursive type. This encoding, however, suffers from the efficiency and abstraction problems mentioned in the introduction.

The only asynchronous modal FRP calculus that we are aware of is $\lambda_{\mathrm{widget}}$ defined by Graulund et al. [2021], which takes $\diamond$ as a type constructor primitive and endows it with synchronisation

primitive similar to select in Async RaTT. However, the programming primitives in λ_{widget} are very different from the ones use here. For example, λ_{widget} allows an element of $\Diamond A$ to be decomposed into a time and an element of A at that time, and much programming with $\Diamond$ uses this decomposition. There is also no delay type constructor $\bigcirc$, so $\textcircled{\exists}$ is not expressible: Unlike $\textcircled{\exists}A$, an element of $\Diamond A$ could give a value of type A already now. Graulund et al. provide a denotational semantics for λ_{widget}, but no operational semantics, and no operational guarantees as proved here.

Another approach to avoiding space leaks and non-causal reactive programs is to devise a carefully designed interface to manipulate signals such as Yampa [Nilsson et al. 2002] or FRPNow! [Ploeg and Claessen 2015]. Rhine [Bärenz and Perez 2018] is a recent refinement of Yampa that annotates signal functions with type-level clocks, which allows the construction of complex dataflow graphs that combine subsystems running at different clock speeds. The typing discipline fixes the clock of each subsystem *statically* at compile time, since the aim of Rhine is provide efficient resampling between subsystems. By contrast, the type-level clocks of Async RaTT are existentially quantified, which allows Async RaTT programs to *dynamically* change the clock of a signal, e.g., by using the *switch* combinator from section 3.2.

Elliott [2009] proposed a *push-pull* implementation of FRP, where signals (which in the tradition of classic FRP [Elliott and Hudak 1997] are called behaviours) are updated at discrete time steps (push), but can also be sampled at any time between such updates (pull). We can represent such push-pull signals in Async RaTT using the type $\mathsf{Sig}\,(\mathsf{Time} \to A)$, i.e., at each tick of the clock we get a new function $\mathsf{Time} \to A$ that describes the time-varying value of the signal until the next tick of the clock.

Futures, first implemented in MuliLisp [Halstead 1985] and now commonly found in many programming languages under different names (promise, async/await, delay, etc.), provide a powerful abstraction to facilitate communication between concurrent computations. A value of type $\mathsf{Future}\,A$ is the promise to deliver a value of type A at some time in the future. For example, a function to read the contents of a file could immediately return a value of type $\mathsf{Future}\,\mathit{Buffer}$ instead of blocking the caller until the file was read into a buffer. Async RaTT can provide a similar interface using the type modality $\textcircled{\exists}$, either directly or by defining Future as a guarded recursive type $\mathsf{Future}\,A \cong A + \textcircled{\exists}(\mathsf{Future}\,A)$ to give Future a monadic interface. Since Async RaTT does not require the set of push-only channels to be finite, we could implement a function that takes a filename f and returns a result of type $\mathsf{Future}\,\mathsf{Buffer}$ simply as a family of channels $\mathsf{readFile}_f :_{\mathsf{p}} \mathit{Buffer}$. The machine would monitor delayed computations for clocks containing these channels, initiate reading the corresponding files in parallel, and provide the value of type *Buffer* on the channel upon completion of the file reading procedure.

As mentioned earlier, Krishnaswami et al. [2012] used a linear typing discipline to obtain static memory bounds. In addition to such memory bounds, synchronous (dataflow) languages such as Esterel [Berry and Cosserat 1985], Lustre [Caspi et al. 1987], and Lucid Synchrone [Pouzet 2006] even provide bounds on runtime. Despite these strong guarantees, Lucid Synchrone affords a high-level, modular programming style with support for higher-order functions. However, to achieve such static guarantees, synchronous dataflow languages must necessarily enforce strict limits on the dynamic behaviour, disallowing both time-varying values of arbitrary types (e.g., we cannot have a stream of streams) and dynamic switching (i.e., no functionality equivalent to the *switch* combinator). Both Lustre and Lucid Synchrone have a notion of a clock, which is simply a stream of Booleans that indicates at each tick of the global clock, whether the local clock ticks as well.

7 CONCLUSION AND FUTURE WORK

This paper presented Async RaTT, the first modal language for asynchronous FRP with operational guarantees. We showed how the new modal type ⋵ for asynchronous delay can be used to annotate the runtime system with dependencies from output channels to input channels, ensuring that outputs are only recomputed when necessary. The examples of the integral and the derivative even show how the programmer can actively influence the update rate of output channels.

The choice of Fitch-style modalities is a question of taste, and we believe that the results could be reproduced in a dual context language. Even though Fitch-style uses non-standard operations on contexts, other languages in the RaTT family have been implemented as libraries in Haskell [Bahr 2022]. We therefore believe that also Async RaTT can be implemented in Haskell or other functional programming languages, giving programmers access to a combination of features from RaTT and the hosting programming language.

One aspect missing from Async RaTT is filtering of output channels. For example, it is not possible to write a filter function that only produces output when some condition on the input is met. The best way to do model this is using an output channel of type $\mathsf{Maybe}(A)$, leaving it to the runtime system to only push values of type A to the consumers of the output channel. This way the filtering is external to the programming language. We see no way to meaningfully extend the runtime model of Async RaTT to internalise it.

ACKNOWLEDGMENTS

Møgelberg was supported by the Independent Research Fund Denmark grant number 2032-00134B.

REFERENCES

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712 00283.

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

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), 1–27. https://doi.org/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, POPL (2021), 1–28.

Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3242744.3242757

Gérard Berry and Laurent Cosserat. 1985. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, DE, 389–448.

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science* 30, 2 (2020), 118–138.

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John 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* (Munich, West Germany) *(POPL '87)*. ACM, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 361–372. https://doi.org/10.1145/2535838.2535881

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, Springer International Publishing, Cham, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM (JACM)* 48, 3 (2001), 555–604.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

Christian Uldal Graulund, Dmitrij Szamozvancev, and Neel Krishnaswami. 2021. Adjoint Reactive GUI Programming.. In *FoSSaCS*. 289–309.

Robert H. Halstead. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

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, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, Koen Claessen and Nikhil Swamy (Eds.). ACM, Philadelphia, PA, USA, 49–60. https://doi.org/10.1145/2103776.2103783

Alan Jeffrey. 2014. Functional Reactive Types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. ACM, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/2603088.2603106

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. https://doi.org/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 (ICFP '13)*. ACM, Boston, Massachusetts, USA, 221–232. https://doi.org/10.1145/2500365.2500588

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 257–266. https://doi.org/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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, PA, USA, 45–58. https://doi.org/10.1145/2103656.2103665

Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. IEEE Computer Society, Washington, DC, USA, 255–266. https://doi.org/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*. ACM, New York, NY, USA, 51–64. https://doi.org/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 (ICFP 2015)*. Association for Computing Machinery, Vancouver, BC, Canada, 302–314. https://doi.org/10.1145/2784731.2784752 00019.

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

Marc Pouzet. 2006. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI* 1 (2006), 25.

A PROOF OF FUNDAMENTAL PROPERTY

Given a heap η we use the following notation to construct a well-formed store with $\langle \kappa \mapsto v \rangle$ as follows:

$$\mathrm{tick}_{\kappa \mapsto v}(\eta) = [\eta]_{\kappa \in} \langle \kappa \mapsto v \rangle [\eta]_{\kappa \notin}$$

Lemma A.1 (Machine monotonicity)**.** *If* $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v; \sigma' \rangle$*, then* $\sigma \sqsubseteq \sigma'$*.*

Proof. Straightforward induction on $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v; \sigma' \rangle$. □

Lemma A.2.

(i) If $\sigma \sqsubseteq^{\Delta}_{\checkmark} \sigma'$*, then* $\mathrm{gc}(\sigma) \sqsubseteq \mathrm{gc}(\sigma')$*.*
(ii) $\mathrm{gc}(\sigma) \sqsubseteq^{\Delta}_{\checkmark} \sigma$*.*
(iii) If $\eta \sqsubseteq \eta'$ *then* $\mathrm{tick}_{\kappa \mapsto v}(\eta) \sqsubseteq \mathrm{tick}_{\kappa \mapsto v}(\eta')$*.*

Proof. By a straightforward case analysis. □

Lemma A.3.

(i) $[\mathrm{tick}_{\kappa \mapsto v}(\eta)]_{\Theta} = \mathrm{tick}_{\kappa \mapsto v}\left([\eta]_{\Theta}\right)$*.*
(ii) $[\mathrm{gc}(\sigma)]_{\Theta} = \mathrm{gc}([\sigma]_{\Theta})$*.*

Proof. By a straightforward case analysis. □

Lemma 5.1. *Let* $n \geq n'$*, and* $\sigma \sqsubseteq^{\Delta}_{\checkmark} \sigma'$*.*

(i) $\mathcal{V}_{\Delta}[\![A]\!](n, \sigma) \subseteq \mathcal{V}_{\Delta}[\![A]\!](n', \sigma')$*.*
(ii) $\mathcal{T}_{\Delta}[\![A]\!](n, \sigma) \subseteq \mathcal{T}_{\Delta}[\![A]\!](n', \sigma')$*.*
(iii) $\mathcal{C}_{\Delta}[\![\Gamma]\!](n, \sigma) \subseteq \mathcal{C}_{\Delta}[\![\Gamma]\!](n', \sigma')$*.*

Proof of Lemma 5.1. (i) and (ii) are proved by a well-founded induction using the same well-founded order that we used to argue that both logical relations are well-defined. (iii) is proved by induction on the length of Γ, and using (i). □

Lemma 5.2.

(i) $\mathcal{V}_{\Delta}[\![A]\!](n, \sigma) \subseteq \mathcal{V}_{\Delta}[\![A]\!](n, \mathrm{gc}(\sigma))$*.*
(ii) $\mathcal{C}_{\Delta}[\![\Gamma]\!](n, \sigma) \subseteq \mathcal{C}_{\Delta}[\![\Gamma]\!](n, \mathrm{gc}(\sigma))$ *if* Γ *is tick-free.*

Proof of lemma 5.2. Both items are proved by induction on the size $|A|$ and on the length of Γ, respectively. □

Lemma 5.3.

(i) If $v \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!](n, \sigma)$*, then* $v \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!](n, \sigma')$*, for any* σ' *with* $[\sigma]_{\mathrm{cl}(v)} = [\sigma']_{\mathrm{cl}(v)}$*.*
(ii) If $\gamma \in \mathcal{C}_{\Delta}[\![\Gamma, \checkmark_{\theta}]\!](n, \sigma)$ *and* $\Theta = |\theta\gamma|$ *then* $\gamma \in \mathcal{C}_{\Delta}[\![\Gamma, \checkmark_{\theta}]\!](n, [\sigma]_{\Theta})$

Proof of Lemma 5.3. Both items are proved by inspection of the definitions of $\mathcal{V}_{\Delta}[\![\circledcirc A]\!](n, \sigma)$ and $\mathcal{C}_{\Delta}[\![\Gamma, \checkmark_{\theta}]\!](n, \sigma)$, respectively. □

The fact that (i) holds for $\mathcal{V}_{\Delta}[\![\circledcirc A]\!](n, \sigma)$ but not for $\mathcal{T}_{\Delta}[\![\circledcirc A]\!](n, \sigma)$ is the reason we needed to restrict the calculus so that adv and select may only be applied to values.

Lemma A.4. $\mathcal{V}_{\Delta}[\![A]\!](n, \sigma) = \mathcal{V}_{\Delta}[\![A]\!](n, \emptyset)$ *for any stable type* A*.*

Proof. By straightforward induction on the size of A. □

Lemma A.5. *Let* $\gamma \in \mathcal{C}_{\Delta}[\![\Gamma, \Gamma']\!](n, \sigma)$ *such that* Γ' *is tick-free. Then* $\gamma|_{\Gamma} \in \mathcal{C}_{\Delta}[\![\Gamma]\!](n, \sigma)$*.*

Proof. By a straightforward induction on the length of Γ'. □

Lemma A.6. *If $\gamma \in C_\Delta[\![\Gamma]\!](n,\sigma)$, then $\gamma|_{\Gamma^\square} \in C_\Delta[\![\Gamma^\square]\!](n,\emptyset)$.*

Proof. By a straightforward induction on the length of Γ and using Lemma A.4. □

Lemma A.7. *If $t \in \mathcal{V}_\Delta[\![A]\!](w)$, then t is a value.*

Proof. By inspection of the definition. □

Lemma A.8. $\mathcal{V}_\Delta[\![A]\!](w) \subseteq \mathcal{T}_\Delta[\![A]\!](w)$.

Proof. Let $t \in \mathcal{V}_\Delta[\![A]\!](n,\sigma)$, and $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\iota : \Delta$. By Lemma A.7, t is a value and thus $\langle t;\sigma'\rangle \Downarrow^\iota \langle t;\sigma'\rangle$. By Lemma 5.1, we have that $t \in \mathcal{V}_\Delta[\![A]\!](n,\sigma')$, which in turn implies $t \in \mathcal{V}_\Delta[\![A]\!](n,\sigma')$ by Lemma 5.1. □

Lemma A.9. *If v is a value with $v \in \mathcal{T}_\Delta[\![A]\!](w)$, then $v \in \mathcal{V}_\Delta[\![A]\!](w)$.*

Proof. Let $v \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$, and pick an arbitrary $\iota : \Delta$. Since $\langle v;\sigma\rangle \Downarrow^\iota \langle v;\sigma\rangle$, we have by definition that $v \in \mathcal{V}_\Delta[\![A]\!](n,\sigma)$. □

Lemma A.10. *If $\Gamma \vdash_\Delta \theta : \mathsf{Clock}$ and $\gamma \in C_\Delta[\![\Gamma]\!](w)$, then $\theta\gamma$ is a closed clock expression and $|\theta\gamma| \subseteq \mathsf{dom_p}(\Delta)$.*

Proof. We proceed by induction on $\Gamma \vdash_\Delta \theta : \mathsf{Clock}$.

- $\dfrac{\Gamma \vdash_\Delta \theta : \mathsf{Clock} \qquad \Gamma \vdash_\Delta \theta' : \mathsf{Clock}}{\Gamma \vdash_\Delta \theta \sqcup \theta' : \mathsf{Clock}}$
  By induction, $\theta\gamma$ and $\theta'\gamma$ are closed and $|\theta\gamma| \subseteq \mathsf{dom}(\Delta)$. Hence, $(\theta \sqcup \theta')\gamma = \theta\gamma \sqcup \theta'\gamma$ is closed and $|(\theta \sqcup \theta')\gamma| = |\theta\gamma| \cup |\theta'\gamma| \subseteq \mathsf{dom}(\Delta)$.
- $\dfrac{\Gamma \vdash_\Delta v : \textcircled{\exists} A}{\Gamma \vdash_\Delta \mathsf{cl}(v) : \mathsf{Clock}}$
  $\Gamma \vdash_\Delta v : \textcircled{\exists} A$ implies that $v\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](w)$ (because either v is a variable or $v = \mathsf{wait}_\kappa$ for some clock κ). Hence, $v\gamma \in \mathsf{Loc}_\Delta \cup \{\mathsf{wait}_\kappa \mid \kappa \in \mathsf{dom_p}(\Delta)\}$ and thus $\mathsf{cl}(v)\,\gamma$ is a closed clock expression and $|\mathsf{cl}(v)\,\gamma| \subseteq \mathsf{dom_p}(\Delta)$.

□

Lemma A.11. *Let A be a value type. Then $v \in \mathcal{V}_\Delta[\![A]\!](n,\sigma)$ iff $\vdash_\Delta v : A$.*

Proof. Straightforward induction on A. □

Lemma A.12. *Let $\eta_N \in \mathsf{Heap}^\kappa$, $v \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n+1,(\eta_N,[\eta_L]_{\kappa\notin}))$, $\iota : \Delta, \vdash w : \Delta(\kappa)$ and $\kappa \in |\mathsf{cl}(v)|$. Then, for any $\sigma \sqsupseteq \eta_N\,\langle \kappa \mapsto w\rangle\,\eta_L$, there are some σ' and $v' \in \mathcal{V}_\Delta[\![A]\!](n,\sigma')$ with $\langle \mathsf{adv}\,v;\sigma\rangle \Downarrow^\iota \langle v';\sigma'\rangle$.*

Proof. By definition, v is either some $l \in \mathsf{Loc}$ or of the form $\mathsf{wait}_{\kappa'}$.

- In the former case, we have by the definition of the value relation and Lemma A.3,
$$(\eta_N,[\eta_L]_{\kappa\notin})(l) \in \mathcal{T}_\Delta[\![A]\!](n,\left[\eta_N\,\langle\kappa\mapsto w\rangle\,[\eta_L]_{\kappa\notin}\right]_{\mathsf{cl}(l)}).$$
In turn, this implies by Lemma 5.1 that
$$(\eta_N,[\eta_L]_{\kappa\notin})(l) \in \mathcal{T}_\Delta[\![A]\!](n,\eta_N\,\langle\kappa\mapsto w\rangle\,\eta_L).$$
Moreover, since $\kappa \in \mathsf{cl}(l)$ we know that $(\eta_N,[\eta_L]_{\kappa\notin})(l) = \eta_N(l)$. Hence, there is a reduction $\langle\eta_N(l);\sigma\rangle \Downarrow^\iota \langle v';\sigma'\rangle$ with $v' \in \mathcal{V}_\Delta[\![A]\!](n,\sigma')$, which by definition means that $\langle \mathsf{adv}\,v;\sigma\rangle \Downarrow^\iota \langle v';\sigma'\rangle$.

- In the latter case, we know that $\kappa' = \kappa$ because $\kappa \in |\mathsf{cl}\,(\mathsf{wait}_{\kappa'})| = \{\kappa'\}$. Moreover, we have that $\kappa :_c A \in \Delta$ for $c \in \{\mathrm{p}, \mathrm{bp}\}$ and thus $\vdash w : A$. By definition, $\eta_N\,\langle \kappa \mapsto w\rangle\,\eta_L \sqsubseteq \sigma$ implies that σ is of the form $\eta'_N\,\langle \kappa \mapsto w\rangle\,\eta'_L$. Hence, by definition $\langle \mathsf{adv}\,\mathsf{wait}_\kappa; \sigma\rangle \Downarrow^\iota \langle w; \sigma\rangle$. Moreover, by Lemma A.11 $w \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$.

□

Theorem 5.4 (Fundamental property)**.** *Given* $\Gamma \vdash_\Delta$, $\Gamma \vdash_\Delta t : A$, *and* $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n, \sigma)$, *then* $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$.

Proof of Theorem 5.4. We proceed by structural induction over the typing derivation $\Gamma \vdash_\Delta t : A$. If $t\gamma$ is a value, it suffices to show that $t\gamma \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$, according to Lemma A.8. In all other cases, to prove $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, we assume some input buffer $\iota : \Delta$ and store $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$, and show that there exists σ'' and v s.t. $\langle t\gamma; \sigma\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ and $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$. By Lemma 5.1 we may assume that $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n, \sigma')$.

- $$\frac{\Gamma' \text{ tick-free or } A \text{ stable} \qquad \Gamma, x : A, \Gamma' \vdash_\Delta}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A}$$

  We show that $x\gamma \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$. If Γ' is tick-free, then $x\gamma \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$ by Lemma A.5 If Γ' is not tick-free, it is of the form $\Gamma_1, \not\checkmark_\theta, \Gamma_2$ and A is stable. By Lemma A.5,

  $$\gamma|_{\Gamma, x:A} \in \mathcal{C}_\Delta[\![\Gamma, x : A]\!](n + 1, \sigma')$$

  for some σ'. Hence, $x\gamma \in \mathcal{V}_\Delta[\![A]\!](n + 1, \sigma')$ and by Lemma 5.1 and Lemma A.4 $x\gamma \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$.

- $$\frac{}{\Gamma \vdash_\Delta \langle\rangle : 1}$$

  Follows immediately by definition.

- $$\frac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \mathsf{let}\, x = s\, \mathsf{in}\, t : B}$$

  By induction, we have $s\gamma \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, which means that $\langle s\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$. By Lemma 5.1 and Lemma A.1, $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n, \sigma'')$ and thus

  $$\gamma[x \mapsto v] \in \mathcal{C}_\Delta[\![\Gamma, x : A]\!](n, \sigma'').$$

  Hence, we may apply the induction hypothesis to obtain $t\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!](n, \sigma'')$. Since all elements in the range of γ are closed terms, $t\gamma[x \mapsto v] = (t\gamma)[v/x]$ and thus $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!](n, \sigma'')$. Consequently, $\langle (t\gamma)[v/x]; \sigma''\rangle \Downarrow^\iota \langle w; \sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!](n, \sigma''')$. By definition, we thus have $\langle (\mathsf{let}\, x = s\, \mathsf{in}\, t)\gamma; \sigma'\rangle \Downarrow^\iota \langle w; \sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!](n, \sigma''')$.

- $$\frac{\Gamma, x : A \vdash_\Delta t : B \qquad \Gamma \text{ tick-free}}{\Gamma \vdash_\Delta \lambda x.t : A \to B}$$

  We show that $\lambda x.t\gamma \in \mathcal{V}_\Delta[\![A \to B]\!](n, \sigma)$. To this end, we assume $\sigma' \sqsupseteq^\Delta_\checkmark \mathrm{gc}(\sigma)$, $n' \le n$, and $v \in \mathcal{V}_\Delta[\![A]\!](n', \sigma')$, with the goal of showing $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!](n', \sigma')$. By Lemma 5.2 and Lemma 5.1, $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n', \sigma')$, and thus, by definition, $\gamma[x \mapsto v] \in \mathcal{C}_\Delta[\![\Gamma, x : A]\!](n', \sigma')$. By induction, we then have that $t\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!](n', \sigma')$. Since all elements in the range of γ are closed terms, $t\gamma[x \mapsto v] = (t\gamma)[v/x]$ and thus $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!](n', \sigma')$.

- $$\frac{\Gamma \vdash_\Delta s : A \to B \qquad \Gamma \vdash_\Delta t : A}{\Gamma \vdash_\Delta s\, t : B}$$

By induction, we have $s\gamma \in \mathcal{T}_\Delta[\![A \to B]\!](n,\sigma)$, which means that $\langle s\gamma;\sigma'\rangle \Downarrow^\iota \langle \lambda x.s';\sigma''\rangle$ for some $\lambda x.s' \in \mathcal{V}_\Delta[\![A \to B]\!](n,\sigma'')$. By induction, we also have $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$. Since by Lemma A.1, $\sigma'' \sqsupseteq^\Delta_\checkmark \sigma$, this means that $\langle t\gamma;\sigma''\rangle \Downarrow^\iota \langle v;\sigma'''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A]\!](n,\sigma''')$. Hence, by definition, $s'[v/x] \in \mathcal{T}_\Delta[\![B]\!](n,\sigma''')$, since $\sigma''' \sqsupseteq^{\Delta'}_\checkmark \mathrm{gc}(\sigma'')$ by Lemma A.2 and Lemma A.1. That means that we have $\langle s[v/x];\sigma'''\rangle \Downarrow^\iota \langle w;\sigma''''\rangle$ for some $w \in \mathcal{V}_\Delta[\![B]\!](n,\sigma'''')$. By definition of the machine, we thus have $\langle (s\gamma)(t\gamma);\sigma'\rangle \Downarrow^\iota \langle w;\sigma''''\rangle$.

- $$\frac{\Gamma \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta t' : B}{\Gamma \vdash_\Delta (t,t') : A \times B}$$

By induction, we have $s\gamma \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$, which means that $\langle s\gamma;\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A]\!](n,\sigma'')$. We also have $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$ by induction, which by Lemma A.1 means that $\langle t\gamma;\sigma''\rangle \Downarrow^\iota \langle v';\sigma'''\rangle$ for some $v' \in \mathcal{V}_\Delta[\![B]\!](n,\sigma''')$. Hence, $\langle (t,t');\sigma'\rangle \Downarrow^\iota \langle (v,v');\sigma'''\rangle$, and by Lemma 5.1, $(v,v') \in \mathcal{V}_\Delta[\![A \times B]\!](n,\sigma''')$.

- $$\frac{\Gamma \vdash_\Delta t : A_1 \times A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \pi_i\, t : A_i}$$

By induction, we have $t\gamma \in \mathcal{T}_\Delta[\![A_1 \times A_2]\!](n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle (v_1,v_2);\sigma''\rangle$ with $v_i \in \mathcal{V}_\Delta[\![A_i]\!](n,\sigma'')$. Moreover, by definition, $\langle \pi_i\, t\gamma;\sigma'\rangle \Downarrow^\iota \langle v_i;\sigma''\rangle$.

- $$\frac{\Gamma \vdash_\Delta t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \mathsf{in}_i\, t : A_1 + A_2}$$

By induction, we have $t\gamma \in \mathcal{T}_\Delta[\![A_i]\!](n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A_i]\!](n,\sigma'')$. Hence, by definition, $\langle \mathsf{in}_i\, t\gamma;\sigma'\rangle \Downarrow^\iota \langle \mathsf{in}_i\, v;\sigma''\rangle$ and $\mathsf{in}_i\, v \in \mathcal{V}_\Delta[\![A_1 + A_2]\!](n,\sigma'')$.

- $$\frac{\Gamma, x : A_i \vdash_\Delta t_i : B \qquad \Gamma \vdash_\Delta t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B}$$

By induction, we have $t\gamma \in \mathcal{T}_\Delta[\![A_1 + A_2]\!](n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle \mathsf{in}_i\, v;\sigma''\rangle$ for some $i \in \{1,2\}$ such that $v \in \mathcal{V}_\Delta[\![A_i]\!](n,\sigma'')$. By Lemma 5.1 and Lemma A.1, $\gamma \in C_\Delta[\![\Gamma]\!](n,\sigma'')$ and thus $\gamma[x \mapsto v] \in C_\Delta[\![\Gamma, x : A_i]\!](n,\sigma'')$. Hence, we may apply the induction hypothesis to obtain $t_i\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!](n,\sigma'')$. Since all elements in the range of γ are closed terms, $t_i\gamma[x \mapsto v] = (t_i\gamma)[v/x]$ and thus $(t_i\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!](n,\sigma'')$. Consequently, $\langle (t_i\gamma)[v/x];\sigma''\rangle \Downarrow^\iota \langle w;\sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!](n,\sigma''')$. By definition, we thus have $\langle (\mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2)\gamma;\sigma'\rangle \Downarrow^\iota \langle w;\sigma'''\rangle$, as well.

- $$\frac{\Gamma, \checkmark_\theta \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta \theta : \mathsf{Clock}}{\Gamma \vdash_\Delta \mathsf{delay}_\theta\, t : \textcircled{\exists} A}$$

By definition of the machine we have that $\left\langle \mathsf{delay}_{\theta\gamma}\, t\gamma;\sigma' \right\rangle \Downarrow^\iota \langle l;\sigma''\rangle$, where $\sigma'' = \sigma', l \mapsto t\gamma$ and $\mathrm{cl}(l) = |\theta\gamma|$. By Lemma A.10, $|\theta\gamma| \subseteq \mathrm{dom_p}(\Delta)$. It remains to be shown that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n,\sigma'')$. For the case where $n = 0$, this follows immediately from the fact that $|\theta\gamma| \subseteq \mathrm{dom_p}(\Delta)$.

Assume that $n = n' + 1$, $\kappa \in \Theta$, and $\vdash v : \Delta(\kappa)$, where $\Theta = |\theta\gamma|$. By Lemma 5.1 and Lemma 5.2, we have that $\gamma \in C_\Delta[\![\Gamma]\!](n' + 1, \mathrm{gc}(\sigma''))$. By definition we thus have that

$$\gamma \in C_\Delta[\![\Gamma, \checkmark_\theta]\!](n', [\mathrm{gc}(\sigma'')]_{\kappa \in} \langle \kappa \mapsto v\rangle\, [\mathrm{gc}(\sigma'')]_{\kappa \notin}) = C_\Delta[\![\Gamma, \checkmark_\theta]\!](n', \mathsf{tick}_{\kappa \mapsto v}\, (\mathrm{gc}(\sigma''))),$$

and thus $\gamma \in C_\Delta[\![\Gamma, \checkmark_\theta]\!](n', [\mathsf{tick}_{\kappa \mapsto v}\, (\mathrm{gc}(\sigma''))]_\Theta)$ according to Lemma 5.3. Hence, we can apply the induction hypothesis to conclude that

$$t\gamma \in \mathcal{T}_\Delta[\![A]\!](n', [\mathsf{tick}_{\kappa \mapsto v}\, (\mathrm{gc}(\sigma''))]_\Theta).$$

Since $\sigma''(l) = t\gamma$, we thus have that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n, \sigma'')$.

- $\dfrac{}{\Gamma \vdash_\Delta \mathsf{never} : \textcircled{\exists} A}$

  According to the definition of the machine, we have $\langle \mathsf{never}; \sigma' \rangle \Downarrow^\iota \langle l; \sigma' \rangle$ with $l = \mathsf{alloc}^\emptyset(\sigma)$. Since $\mathsf{cl}(l) = \emptyset$, we know that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n, \sigma')$.

- $\dfrac{\kappa :_c A \in \Delta, c \in \{\mathrm{p}, \mathrm{bp}\}}{\Gamma \vdash_\Delta \mathsf{wait}_\kappa : \textcircled{\exists} A}$

  $\mathsf{wait}_\kappa \gamma = \mathsf{wait}_\kappa \in \mathcal{V}_\Delta[\![A]\!](n, \sigma)$ follows immediately by definition and the premise.

- $\dfrac{\kappa :_c A \in \Delta, c \in \{\mathrm{b}, \mathrm{bp}\}}{\Gamma \vdash_\Delta \mathsf{read}_\kappa : A}$

  Since $\iota : \Delta$, we know that $\vdash \iota(\kappa) : A$. Hence, By definition of the machine $\langle \mathsf{read}_\kappa; \sigma' \rangle \Downarrow^\iota \langle \iota(\kappa); \sigma' \rangle$. Moreover, by Lemma A.11 $\iota(\kappa) \in \mathcal{V}_\Delta[\![A]\!](n, \sigma')$.

- $\dfrac{\Gamma \vdash_\Delta v : \textcircled{\exists} A \qquad \Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma' \vdash_\Delta \mathsf{adv}\, v : A}$

  By Lemma 5.1 we have that $\gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_{\mathsf{cl}(v)}, \Gamma']\!](n, \sigma')$ and by Lemma A.5, $\gamma|_\Gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_{\mathsf{cl}(v)}]\!](n, \sigma')$. Let $\Theta = |\mathsf{cl}(v)\, \gamma|_\Gamma|$. By definition of the context relation, Θ is well-defined and a subset of $\mathrm{dom}_\mathrm{p}(\Delta)$. By definition of the context relation we also find $\kappa \in \Theta$, $\eta_N, \eta_L, \eta'_N, \eta'_L$ such that $\sigma' = \eta_N \langle \kappa \mapsto w \rangle \eta_L$, $\vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = \left[\eta'_N\right]_\Theta$, $[\eta_L]_\Theta = \left[\eta'_L\right]_\Theta$, and $\gamma|_\Gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$. By induction, we thus have that $v\gamma \in \mathcal{T}_\Delta[\![\textcircled{\exists} A]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$. Since $v\gamma$ is a value we have $v\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$ by Lemma A.9. By Lemma 5.3 we then have that

$$v\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!](n+1, (\eta_N, [\eta_L]_{\kappa \notin})).$$

  By Lemma A.12, we then find a reduction $\langle \mathsf{adv}\, v\gamma; \sigma' \rangle \Downarrow^\iota \langle v'; \sigma'' \rangle$ with $v' \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$.

- $\dfrac{\Gamma \vdash_\Delta v_1 : \textcircled{\exists} A_1 \qquad \Gamma \vdash_\Delta v_2 : \textcircled{\exists} A_2 \qquad \vdash \theta = \mathsf{cl}(v_1) \sqcup \mathsf{cl}(v_2) \qquad \Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta \mathsf{select}\, v_1\, v_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)}$

  By Lemma 5.1 we have that $\gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta, \Gamma']\!](n, \sigma')$ and by Lemma A.5, $\gamma|_\Gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta]\!](n, \sigma')$. Let $\Theta_1 = |\mathsf{cl}(v_1)\, \gamma|_\Gamma|$, $\Theta_2 = |\mathsf{cl}(v_2)\, \gamma|_\Gamma|$, and $\Theta = \Theta_1 \cup \Theta_2$. According to the definition of the context relation, Θ is well-defined and a subset of $\mathrm{dom}(\Delta)$. By definition of the context relation we also find $\kappa \in \Theta, \eta_N, \eta_L, \eta'_N, \eta'_L$ such that $\sigma' = \eta_N \langle \kappa \mapsto w \rangle \eta_L, \vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = \left[\eta'_N\right]_\Theta$, $[\eta_L]_\Theta = \left[\eta'_L\right]_\Theta$, and $\gamma|_\Gamma \in \mathcal{C}_\Delta[\![\Gamma]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$. By induction hypothesis, we thus have that $v_i\gamma \in \mathcal{T}_\Delta[\![\textcircled{\exists} A_i]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$ for all $i \in \{1, 2\}$. Since $v_i\gamma$ are values, we also have that $v_i\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_i]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa \notin}))$ by Lemma A.9. By Lemma 5.3 we then have that $v_i\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_i]\!](n+1, (\eta_N, [\eta_L]_{\kappa \notin}))$ for all $i \in \{1, 2\}$. There are two cases to consider:

  – Let $i \in \{1, 2\}$ and $j = 3 - i$ such that $\kappa \in \Theta_i \setminus \Theta_j$:
    By Lemma A.12, there is a reduction $\langle \mathsf{adv}\, v_i\gamma; \sigma' \rangle \Downarrow^\iota \langle u_i; \sigma'' \rangle$ with $u_i \in \mathcal{V}_\Delta[\![A_i]\!](n, \sigma'')$, which by definition means that $\langle \mathsf{select}\, v_1\gamma\, v_2\gamma; \sigma' \rangle \Downarrow^\iota \langle \mathsf{in}_1(\mathsf{in}_i(u_1, u_2)); \sigma'' \rangle$ with $u_j = v_j\gamma$. It thus remains to be shown that $v_j\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n, \sigma'')$. There are two cases to consider.
    * Let $v_j\gamma = l$ for some $l \in \mathrm{Loc}_\Delta$. From $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n+1, (\eta_N, [\eta_L]_{\kappa \notin}))$ and the fact that $\kappa \notin \Theta_j$, we obtain that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n+1, [\eta_L]_{\kappa \notin})$ by using Lemma 5.3. In particular, we use the fact that $[\eta_N]_{\Theta_j} = \emptyset$ since $\eta_N \in \mathsf{Heap}^\kappa$ and $\kappa \notin \Theta_j$. Since $[\eta_L]_{\kappa \notin} \sqsubseteq^\Delta_\checkmark \sigma'$ and, by Lemma A.1, $\sigma' \sqsubseteq^\Delta_\checkmark \sigma''$, we can then use Lemma 5.1 to conclude that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n, \sigma'')$.

  - Let $v_j\gamma = \mathsf{wait}_{\kappa'}$ for some clock κ'. But then $\Gamma \vdash_\Delta v_j : \textcircled{\exists} A_j$ is due to $\kappa' :_{\mathsf{p}} A_j \in \Delta$ or $\kappa' :_{\mathsf{bp}} A_j \in \Delta$ and thus $v_j\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n, \sigma'')$ follows immediately by definition of the value relation.
- $\kappa \in \Theta_1 \cap \Theta_2$: By Lemma A.12, we obtain a reduction $\langle \mathsf{adv}\, v_1\gamma); \sigma'\rangle \Downarrow^\iota \langle v_1'; \sigma''\rangle$ with $v_1' \in \mathcal{V}_\Delta[\![A_1]\!](n, \sigma'')$, and a reduction $\langle \mathsf{adv}\, v_2\gamma; \sigma''\rangle \Downarrow^\iota \langle v_2'; \sigma'''\rangle$ with $v_2' \in \mathcal{V}_\Delta[\![A_2]\!](n, \sigma''')$. By definition we thus obtain a reduction $\langle \mathsf{select}\,(v_1\gamma)\,(v_2\gamma); \sigma'\rangle \Downarrow^\iota \langle \mathsf{in}_2((v_1', v_2')); \sigma'''\rangle$. Moreover, applying Lemma A.1 and Lemma 5.1, we obtain that $v_1' \in \mathcal{V}_\Delta[\![A_1]\!](n, \sigma''')$, which means that we have $\mathsf{in}_2((v_1', v_2')) \in \mathcal{V}_\Delta[\![A_1 \times A_2]\!](n, \sigma''')$.

- $\dfrac{}{\Gamma \vdash_\Delta 0 : \mathsf{Nat}}$
  $0\gamma \in \mathcal{V}_\Delta[\![\mathsf{Nat}]\!](n, \sigma)$ follows immediately by definition.
- $\dfrac{\Gamma \vdash_\Delta t : \mathsf{Nat}}{\Gamma \vdash_\Delta \mathsf{suc}\, t : \mathsf{Nat}}$
  By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![\mathsf{Nat}]\!](n, \sigma)$, which means that $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{suc}^m\, 0; \sigma''\rangle$ for some $m \in \mathbb{N}$. Hence, by definition, $\langle \mathsf{suc}\, t\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{suc}^{m+1}\, 0; \sigma''\rangle$ and $\mathsf{suc}^{m+1}\, 0 \in \mathcal{V}_\Delta[\![\mathsf{Nat}]\!](n, \sigma'')$.
- $\dfrac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : \mathsf{Nat}, y : A \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta u : \mathsf{Nat}}{\Gamma \vdash_\Delta \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, u) : A}$
  We claim that the following holds:

$$\mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, \mathsf{suc}^k\, 0) \in \mathcal{T}_\Delta[\![A]\!](n, \sigma) \text{ for all } k \in \mathbb{N} \tag{5}$$

  To show that $\mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, u\gamma) \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, assume some $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\iota : \Delta$. By induction hypothesis $u\gamma \in \mathcal{T}_\Delta[\![\mathsf{Nat}]\!](n, \sigma)$, which means that $\langle u\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{suc}^m\, 0; \sigma''\rangle$. By (5) and Lemma A.1 we have that $\langle \mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, \mathsf{suc}^m\, 0); \sigma''\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$ with $v \in \mathcal{T}_\Delta[\![A]\!](n, \sigma''')$. By definition of the machine, we also have that $\langle \mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, u\gamma); \sigma'\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$.
  We conclude by showing (5) by induction on k.
  - Case $k = 0$: Let $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\iota : \Delta$. By definition, $\langle 0; \sigma'\rangle \Downarrow^\iota \langle 0; \sigma'\rangle$. By induction hypothesis $s\gamma \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, which means that $\langle s\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$. By definition, $\langle \mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, 0); \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ follows.
  - Case $k = l{+}1$. Let $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\iota : \Delta$. By definition, $\langle \mathsf{suc}^k\, 0; \sigma'\rangle \Downarrow^\iota \langle \mathsf{suc}(\mathsf{suc}^l\, 0); \sigma'\rangle$. By induction (on k), we have that $\langle \mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, \mathsf{suc}^l\, 0); \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$. By Lemma A.1 and Lemma 5.1, we have $\gamma \in C_\Delta[\![\Gamma]\!](n, \sigma'')$ and thus

$$\gamma[x \mapsto \mathsf{suc}^l\, 0, y \mapsto v] \in C_\Delta[\![\Gamma, x : \mathsf{Nat}, y : A]\!](n, \sigma'').$$

    By induction we thus obtain that

$$(t\gamma)[\mathsf{suc}^l\, 0/x, v/y] = t\gamma[x \mapsto \mathsf{suc}^l\, 0, y \mapsto v] \in \mathcal{T}_\Delta[\![A]\!](n, \sigma''),$$

    which means that there is a reduction $\langle (t\gamma)[\mathsf{suc}^l\, 0/x, v/y]; \sigma''\rangle \Downarrow^\iota \langle w; \sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![A]\!](n, \sigma''')$. According to the definition of the machine, we thus have

$$\langle \mathsf{rec}_{\mathsf{Nat}}(s\gamma, x\, y.t\gamma, \mathsf{suc}^k\, 0); \sigma'\rangle \Downarrow^\iota \langle w; \sigma'''\rangle\,.$$

- $\dfrac{\Gamma^\square, x : \textcircled{\forall} A \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{fix}\, x.t : A}$
  We will show that

$$\mathsf{dfix}\, x.t\gamma \in \mathcal{V}_\Delta[\![\textcircled{\forall} A]\!](m, \emptyset) \text{ for all } m \le n. \tag{6}$$

Using (6), Lemma A.6, and Lemma 5.1, we then obtain that $(\gamma|_{\Gamma^\square})[x \mapsto \mathsf{dfix}\, x.t\gamma] \in \mathcal{C}_\Delta[\![\Gamma^\square, x : \circledcirc A]\!](n, \sigma)$. Hence, by induction, $t[\mathsf{dfix}\, x.t/x]\gamma = t(\gamma|_{\Gamma^\square})[x \mapsto \mathsf{dfix}\, x.t\gamma] \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$, which means that we find $\langle t[\mathsf{dfix}\, x.t/x]\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$. By definition of the machine we thus obtain the desired $\langle \mathsf{fix}\, x.t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$.
We prove (6) by induction on m.
If $m = 0$, then (6) follows immediately from the fact that $\mathsf{dfix}\, x.t\gamma$ is a closed term.
Let $m = m' + 1$. By Lemma 5.1 and Lemma A.6, $\gamma|_{\Gamma^\square} \in \mathcal{C}_\Delta[\![\Gamma^\square]\!](m', \emptyset)$. By the induction hypothesis (on (6)) we have $\mathsf{dfix}\, x.t\gamma \in \mathcal{V}_\Delta[\![\circledcirc A]\!](m', \emptyset)$ and thus

$$(\gamma|_{\Gamma^\square})[x \mapsto \mathsf{dfix}\, x.t\gamma] \in \mathcal{C}_\Delta[\![\Gamma^\square, x : \circledcirc A]\!](m', \emptyset).$$

Hence, by induction, $t[\mathsf{dfix}\, x.t/x]\gamma = t(\gamma|_{\Gamma^\square})[x \mapsto \mathsf{dfix}\, x.t\gamma] \in \mathcal{T}_\Delta[\![A]\!](m', \emptyset)$, which allows us to conclude that $\mathsf{dfix}\, x.t\gamma \in \mathcal{V}_\Delta[\![\circledcirc A]\!](m, \emptyset)$.

- $\dfrac{\Gamma \vdash_\Delta x : \circledcirc A \qquad \Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta}{\Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta \mathsf{adv}\, x : A}$
  By Lemma 5.1 we have that $\gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta, \Gamma']\!](n, \sigma')$ and by Lemma A.5, $\gamma|_\Gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta]\!](n, \sigma')$. Let $\Theta = |\theta\gamma|_\Gamma|$. According the definition of the context relation, Θ is well-defined and we find $\kappa \in \Theta, \vdash w : \Delta(\kappa)$, $\eta_N, \eta_L, \eta'_N, \eta'_L$ such that $\sigma' = \eta_N\, \langle \kappa \mapsto w\rangle\, \eta_L$, $[\eta_N]_\Theta = \left[\eta'_N\right]_\Theta$, $[\eta_L]_\Theta = \left[\eta'_L\right]_\Theta$, and $\gamma|_\Gamma \in \mathcal{C}_{\Delta'}[\![\Gamma]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa\notin}))$. By Lemma A.5, we thus have that $\gamma(x) = \mathsf{dfix}\, y.t$ with $\mathsf{dfix}\, y.t \in \mathcal{V}_\Delta[\![\circledcirc A]\!](n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa\notin}))$. By definition, this implies that $t[\mathsf{dfix}\, y.t/y] \in \mathcal{T}_\Delta[\![A]\!](n, \emptyset)$. That is, we find a reduction $\langle t[\mathsf{dfix}\, y.t/y]; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma'')$, which by definition means that we also have a reduction $\langle \mathsf{adv}\, x\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$.
- $\dfrac{\Gamma^\square \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{box}\, t : \square A}$
  We show that $\mathsf{box}\, t\gamma \in \mathcal{V}_\Delta[\![\square A]\!](n, \sigma)$. By Lemma A.6, $\gamma|_{\Gamma^\square} \in \mathcal{C}_\Delta[\![\Gamma^\square]\!](n, \emptyset)$. Hence, by induction, $t\gamma = t\gamma|_{\Gamma^\square} \in \mathcal{T}_\Delta[\![A]\!](n, \emptyset)$, and thus $\mathsf{box}\, t\gamma \in \mathcal{V}_\Delta[\![\square A]\!](n, \sigma)$.
- $\dfrac{\Gamma \vdash_\Delta t : \square A}{\Gamma \vdash_\Delta \mathsf{unbox}\, t : A}$
  By induction hypothesis, we have that $t\gamma \in \mathcal{T}_\Delta[\![\square A]\!](n, \sigma)$. That is, $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{box}\, s; \sigma''\rangle$ for some $s \in \mathcal{T}_\Delta[\![A]\!](n, \emptyset)$. Hence, $\langle s; \sigma''\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$ such that $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma''')$ which, by Lemma 5.1, implies $v \in \mathcal{V}_\Delta[\![A]\!](n, \sigma''')$. Moreover, by definition of the machine we have that $\langle \mathsf{unbox}\, t; \sigma'\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$.
- $\dfrac{\Gamma \vdash_\Delta t : \mathsf{Fix}\, \alpha.A}{\Gamma \vdash_\Delta \mathsf{out}\, t : A[\circledast(\mathsf{Fix}\, \alpha.A)/\alpha]}$
  By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![\mathsf{Fix}\, \alpha.A]\!](n, \sigma)$, which means that $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{into}\, v; \sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A[\circledast(\mathsf{Fix}\, \alpha.A)/\alpha]]\!](n, \sigma'')$. Moreover, by definition of the machine we consequently have $\langle \mathsf{out}\, t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$.
- $\dfrac{\Gamma \vdash_\Delta t : A[\circledast(\mathsf{Fix}\, \alpha.A)/\alpha]}{\Gamma \vdash_\Delta \mathsf{into}\, t : \mathsf{Fix}\, \alpha.A}$
  By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![A[\circledast(\mathsf{Fix}\, \alpha.A)/\alpha]]\!](n, \sigma)$, which means that $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A[\circledast(\mathsf{Fix}\, \alpha.A)/\alpha]]\!](n, \sigma'')$. Hence, by definition, $\langle \mathsf{into}\, t\gamma; \sigma'\rangle \Downarrow^\iota \langle \mathsf{into}\, v; \sigma''\rangle$ and $\mathsf{into}\, v \in \mathcal{V}_\Delta[\![\mathsf{Fix}\, \alpha.A]\!](n, \sigma'')$.

□

Received 2023-03-01; accepted 2023-06-27