Asynchronous modal FRP: Ticking all the boxes

PATRICK BAHR

IT University of Copenhagen, Denmark

RASMUS EJLERS MØGELBERG

IT University of Copenhagen, Denmark

Abstract

Over the past fifteen years, several languages for functional reactive programming (FRP) have been suggested that 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 dynamically during execution, upper bounds on these can be determined statically by the type system.

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 goal of functional reactive programming (FRP), originally proposed by Elliott and Hudak (1997), is to provide programmers with the right abstractions to write reactive programs in a functional style, allowing for concise, modular programs, as well as modular reasoning principles for them. For such abstractions to be useful, it is important that they are designed so that efficient low-level implementations may be algorithmically generated from high-level 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 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

Journal of Functional Programming, Volume 36, Article 13 DOI: 10.46298/jfp.17791
Submitted July 2025; accepted August 2026

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.

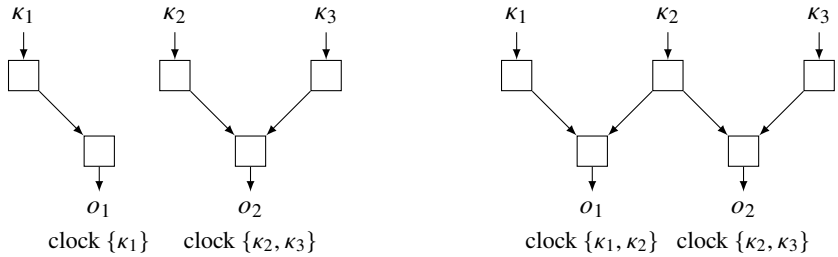
These requirements have led to the development of modal FRP (Bahr 2022; Bahr et al. 2019; Jeffrey 2012, 2014; Jeltsch 2012; Krishnaswami 2013; Krishnaswami and Benton 2011; Krishnaswami et al. 2012), a family of languages using modal types to ensure that all programs can be implemented efficiently. The most important modal type constructor is the later modality $\bigcirc$, 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 $Sig\ A \cong A \times \bigcirc(Sig\ A)$ 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 enforces 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 over time 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 a signal of type *Maybe Char*. In the modal FRP languages of Bahr et al. (2019) and 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



(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.

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 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 $\Box$, pronounced "box", that is used to classify stable data, as well as two new modalities to classify delayed computations: $\textcircled{\exists}$, pronounced "later", to classify data that arrives at *some time* in the future, and $\textcircled{\forall}$, pronounced "next", to classify data that is available at *any time* in the future. 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 next 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 as a recursive type $\mathit{Sig}\ A \cong A \times \textcircled{\exists}(\mathit{Sig}\ A)$. That means, a signal consists of its current value of type A along with a tail of type $\textcircled{\exists}(\mathit{Sig}\ A)$ that is a delayed computation producing future values of the signal. Since this delayed computation uses the $\textcircled{\exists}$ modality, it contains a clock that specifies when the next value of the signal becomes available. Consequently, as the signal unfolds over time, the clock associated with its tail may change from one time 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 its delayed argument 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

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

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 is received on channel $\kappa \in \theta_1 \setminus \theta_2$, only the first delayed computation is evaluated. The *sync* operator can be used to implement signal combinators that dynamically update the dataflow

graph of a program. An example of such a signal combinator is

$$switch : Sig\ A \to \Diamond(Sig\ A) \to Sig\ A$$

The signal *switch xs ys* first behaves like the signal *xs*, but switches to *ys* as soon as it arrives, namely when its clock ticks.

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 runtime environment to schedule the order in which inputs are processed. What we mean by asynchronicity is that output channels may be updated independently and at different times. This is reflected in the type system by $\Diamond$ not being an applicative functor as explained above.

The modal type $\bigcirc A$ 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

$$\Box(\bigcirc A \to A) \to A$$

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

The difference between the $\Diamond$ and $\bigcirc$ modalities is that the timing of the former is determined by external events, whereas the timing of the latter is determined by the program itself. That is, a value $v : \Diamond A$ waits for an external process to provide data so that the program can produce a value of type A, whereas a value $v : \bigcirc A$ is a suspended computation that is independent of any external event and can thus be resumed to produce a value of type A at any time in the future.

1.2 Operational semantics and results

We present an operational semantics that maps 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 machine. This semantics takes a term, a store, and values received on input channels and produces a value and an updated store. The store contains delayed computations stored in up to two separate heaps: one heap contains previously stored delayed computations that the evaluation semantics can now run, and the other heap can be used to store *new* delayed computations to be evaluated at a later step. The second part of the operational semantics is the *reactive semantics*, which describes the machine which, at each step, locates the output signals to be updated and executes the corresponding delayed computations according to the evaluation semantics in order 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 on the same channel arrives, depending on the kind of the input channel. Our metatheoretic 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.

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 received on channels in $\Delta \setminus \Delta'$. Note that this signal independence result holds despite the ability to express combinators like *switch*, which dynamically change the dataflow graph of a program.

1.3 Overview

The paper is organised as follows: Section 2 presents the syntax and type system of *Async RaTT*. Section 3 demonstrates the expressiveness 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. Section 4 defines the operational semantics, illustrates it with an example, and presents the main results. Section 5 presents the proofs of the main results, and in particular defines the Kripke logical relation used for the proofs. Section 6 presents a more general type system called *Full Async RaTT* that dispenses with some of the limitations of *Async RaTT*. Moreover, we show that closed *Full Async RaTT* terms can be algorithmically transformed into *Async RaTT* terms of the same type. Finally, section 7 and section 8 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.

1.4 Additional material

This paper extends the conference paper (Bahr and Møgelberg 2023) with the following additional contributions:

- The type system of *Async RaTT*, presented in section 2, has been generalised and simplified. It now allows arbitrarily many ticks in the typing context and places no restrictions on the ticks occurring in the typing context when typing lambda abstractions.
- We include an additional example program in section 3.4.
- We have revised the proofs of the metatheoretic results in section 5: We made a small change to the Kripke logical relation to accommodate the new type system of *Async RaTT*; we adapted the proofs to the new type system as well as the revised Kripke logical relation; and we expanded some proofs to make them less terse.
- Section 6 is entirely new: We present a type system, called *Full Async RaTT*, which further generalises *Async RaTT*, and we show that closed *Full Async RaTT* programs can be transformed into *Async RaTT* programs of the same type. In addition, we give examples demonstrating that *Full Async RaTT* programs do not in general satisfy the operational properties of *Async RaTT*, and we illustrate how the program transformation from *Full Async RaTT* to *Async RaTT* recovers these properties.

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

Fig. 2. Syntax.

2 Async RaTT

In this section, we give an overview of *Async RaTT*, referring to Figures 2 and 3 for the full specification of its syntax and typing rules.

An *Async RaTT* program has access to a set of input channels, 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

$$\mathit{keyPressed} :_p \mathit{Nat}, \mathit{mouseCoord} :_{bp} \mathit{Nat} \times \mathit{Nat}, \mathit{time} :_b \mathit{Float} \tag{1}$$

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 input context above, we want the program to react to user keypresses immediately, and so updates to this channel 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 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 input channels are assumed to have value types, i.e., any declaration $\kappa :_c A$ in Δ must have a value type A. Intuitively speaking, value types classify basic values that contain no

$$\frac{}{\cdot \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad x \notin dom(\Gamma)}{\Gamma, x : A \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \qquad \Gamma \vdash_\Delta \theta : Clock}{\Gamma, \checkmark_\theta \vdash_\Delta}$$

$$\frac{\Gamma \vdash_\Delta \theta : Clock \qquad \Gamma \vdash_\Delta \theta' : Clock}{\Gamma \vdash_\Delta \theta \sqcup \theta' : Clock} \qquad \frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A}{\Gamma \vdash_\Delta cl(v) : Clock}\ \star \qquad \frac{\Gamma' \text{ tick-free or } A \text{ stable}}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A}$$

$$\frac{}{\Gamma \vdash_\Delta () : 1} \qquad \frac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta let\, x = s\, in\, t : B} \qquad \frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash_\Delta t : A \to B \qquad \Gamma \vdash_\Delta t' : A}{\Gamma \vdash_\Delta t\, t' : B} \qquad \frac{\Gamma \vdash_\Delta t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta in_i\, t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash_\Delta t_i : B \qquad \Gamma \vdash_\Delta t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta case\, t\, of\, in_1\, x.t_1; in_2\, x.t_2 : B} \qquad \frac{\Gamma \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta t' : B}{\Gamma \vdash_\Delta (t, t') : A \times B}$$

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

$$\frac{\kappa :_c A \in \Delta \qquad c \in \{p, bp\}}{\Gamma \vdash_\Delta wait_\kappa : \textcircled{\exists} A} \qquad \frac{\kappa :_c A \in \Delta \qquad c \in \{b, bp\}}{\Gamma \vdash_\Delta read_\kappa : A} \qquad \frac{\Gamma \vdash_\Delta v : \textcircled{\exists} A \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_{cl(v)}, \Gamma' \vdash_\Delta adv\, v : A}\ \star$$

$$\frac{\Gamma \vdash_\Delta v_1 : \textcircled{\exists} A_1 \qquad \Gamma \vdash_\Delta v_2 : \textcircled{\exists} A_2 \qquad \theta = cl(v_1) \sqcup cl(v_2) \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta select\, v_1\, v_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)}\ \star \qquad \frac{}{\Gamma \vdash_\Delta 0 : Nat}$$

$$\frac{\Gamma \vdash_\Delta t : Nat}{\Gamma \vdash_\Delta suc\, t : Nat} \qquad \frac{\Gamma \vdash_\Delta s : A \qquad \Gamma, x : Nat, y : A \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta u : Nat}{\Gamma \vdash_\Delta rec_{Nat}(s, x\, y.t, u) : A}$$

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

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

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

Fig. 3. Typing rules for *Async RaTT*. Rules marked with ⋆ will be generalised in section 6.

delayed computations and thus exclude function types and all modal types. The grammar for value types is given in Figure 2.

2.1 Clocks and the later modality Ⓔ

A clock is effectively a set of push channels (i.e., *p* or *bp*), which the program may have to react to. For instance, ∅, {*keyPressed*} and {*keyPressed*, *mouseCoord*} are all examples of

clocks for the example input context given in (1) above. The type $\Diamond\!\!\!-A$ is a type of delayed computation on an existentially quantified clock. In other words, a value of type $\Diamond\!\!\!-A$ 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 θ is $\{keyPressed, mouseCoord\}$, then the data of type A can be computed once *keyPressed* or *mouseCoord* receives new input. On the other hand, if the associated clock θ is the empty clock $\emptyset$, then no event can lead θ to tick, and thus the data of type A will never be computed.

Since types of the form $\Diamond\!\!\!-A$ are existential types, one can obtain the clock $cl(v)$ for any *value* v of these types. The values of type $\Diamond\!\!\!-A$ are variables and $wait_\kappa$, where κ is a push channel. The latter acts as a reference to the next value pushed on κ, and so we can intuitively think of the clock expression $cl(wait_\kappa)$ as representing the clock $\{\kappa\}$. 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 $\Diamond\!\!\!-$, this means that introduction and elimination rules use a special symbol $\checkmark_\theta$, referred to as a *tick*, in the typing context. One can think of a tick $\checkmark_\theta$ as representing a tick of the clock θ. Thus, $\checkmark_\theta$ divides the judgement into two parts. On the one side we have the elements that are available before the tick, namely the variables to the left of $\checkmark_\theta$, and on the other side we have the elements that are available after the tick, namely the variables to the right of $\checkmark_\theta$ as well as the term t. For example, we can interpret the judgement $x : A, \checkmark_\theta, y : B \vdash_\Delta t : C$ as saying that if x is available before the tick of the clock θ and y is available after the tick of θ, then t is available after the tick of θ. A typing context may contain several ticks, each of which marks the passage of one time step on a particular clock. For example, the typing judgement $x : A, \checkmark_\theta, y : B, \checkmark_{\theta'} \vdash_\Delta t : C$ states that t is available after both θ and (subsequently) θ' have ticked, and that the variable y arrived after θ ticked but before the subsequent tick of θ'.

With this intuition of clocks and ticks in mind, the elimination rule for $\Diamond\!\!\!-$ should be read as follows: If v has type $\Diamond\!\!\!-A$ now, then $adv\, v$ has type A after a tick on the clock $cl(v)$. Similarly, the introduction rule for $\Diamond\!\!\!-$ should be read as follows: If t has type A after a tick on clock θ, then $delay_\theta\, t$ has type $\Diamond\!\!\!-A$ now.

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

Two delayed values $v_1 : \Diamond\!\!\!-A_1$ and $v_2 : \Diamond\!\!\!-A_2$ can be synchronised using *select* once a tick on the union clock $cl(v_1) \sqcup cl(v_2)$ has occurred. The type of $select\, v_1\, v_2$ reflects the three possible cases for such a tick: It could be on exactly one of the two clocks $cl(v_1)$ and $cl(v_2)$, or it could be on both clocks. For example, if the tick is on $cl(v_1)$ but not on $cl(v_2)$, then data of type $A_1 \times \Diamond\!\!\!-A_2$ can be computed. The *sync* operator shown in section 1.1 can be defined using *select*:

$$sync = \lambda x.\lambda y.delay_{cl(x)\sqcup cl(y)}\ (select\, x\, y)$$

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. This behaviour matches the original `select` primitive in *Concurrent ML* (Reppy 1999). By contrast, in *Async RaTT*, providing all three cases is crucial for the operational results of section 4.

Note that the typing rules allow us to apply *select* and *adv* only to values. As we will show in section 6.3, this restriction is necessary to ensure that the operational semantics, and in particular its aggressive garbage collection strategy, is sound. This restriction also means that clock expressions are always built from values and thus do not need to be evaluated. For example, evaluating $delay_{cl(t)}\,(adv\,t)$ would require evaluating t twice: first to evaluate the clock, and then to evaluate the term itself. Elimination of $\Circledexists$ can be done for general terms t using a combination of let-binding and *adv*, so that we write $let\,x = t\,in\,delay_{cl(x)}\,(adv\,x)$ instead of $delay_{cl(t)}\,(adv\,t)$. Section 6 presents a proof that this idea works in general: Any closed term that is typeable in a more relaxed type system that allows general terms as arguments to *adv* and *select* can be systematically transformed in a type-preserving way so that it satisfies the stricter typing rules in Figure 3. This transformation is also the basis for the implementation of *Async RaTT* as an embedded language in Haskell (Bahr et al. 2024a).

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 from being 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. This is reflected in the typing rule for variable introduction, which prevents general variables from being introduced across ticks.

For some types, however, values cannot contain such references. We refer to these types as *stable types* and the grammar defining them is given in Figure 2. The two kinds of types that are explicitly not stable are delayed types $\Circledexists A$ and function types $A \to B$. Intuitively speaking, a value v of type $\Circledexists A$ is only available until its clock $cl(v)$ ticks and thus cannot be considered stable. Similarly, a value of type $A \to B$ is a closure which may contain arbitrary data, in particular values of type $\Circledexists A$.

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 $box\,t$ that can be evaluated at any time in the future. This requires t to be typed in a stable context, and so the premise of the typing rule removes all ticks and all variables not of stable type from the context. However, since both $wait_\kappa$ and $read_\kappa$ are typeable in an empty context, they are stable in the sense that $\vdash_\Delta box\,wait_\kappa : \Box(\Circledexists A)$ for any $\kappa :_c A \in \Delta$ where $c \in \{p, bp\}$ and $\vdash_\Delta box\,read_\kappa : \Box A$ for any $\kappa :_c A \in \Delta$ where $c \in \{b, bp\}$.

The $\Box$ modality has a counit and a comultiplication:

$$counit : \Box A \to A \qquad comult : \Box A \to \Box(\Box A)$$
$$counit = \lambda x.unbox\,x \qquad comult = \lambda x.box\,x$$

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 performed during later time steps. More precisely, the

recursion variable x in $\mathit{fix}\, x.t$ has type $\textcircled{\forall} A$, which means that the recursive definition can be unfolded to produce a term of type A at 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 typeable 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 typeable in a stable context.

The type system of *Async RaTT* does not feature an introduction form for the $\textcircled{\forall}$ modality. Since the purpose of $\textcircled{\forall}$ is to facilitate guarded recursion via *fix*, *Async RaTT* only needs to provide a way to eliminate $\textcircled{\forall}$. However, as we will see in section 4.1, there is in fact a *semantic* introduction form *dfix*, which we could give the following typing rule:

$$\frac{\Gamma^{\square}, x : \textcircled{\forall} A \vdash_{\Delta} t : A}{\Gamma \vdash_{\Delta} \mathit{dfix}\, x.t : \textcircled{\forall} A}$$

The soundness proof for the guarded fixed point combinator *fix* does indeed reduce to the soundness of the above typing rule. However, since *dfix* does not serve any purpose apart from facilitating the operational semantics of *fix*, we do not include this typing rule in the calculus.

To understand the difference between the two delay modalities $\textcircled{\exists}$ and $\textcircled{\forall}$ it is instructive to conceptualise them in terms of a more basic type modality $\bigcirc^{\theta}$ that classifies computations that are delayed with respect to a specific clock θ. Assuming such a type modality, we can think of $\textcircled{\exists} A$ as the existential type $\exists\theta.\bigcirc^{\theta} A$ and of $\textcircled{\forall} A$ as the polymorphic type $\forall\theta.\bigcirc^{\theta} A$. A value of the former type consists of a clock θ and a delayed computation that can be performed as soon as θ ticks, while a value of the latter type is a delayed computation that can be performed as soon as *any* clock ticks.

The types $\mathit{Fix}\,\alpha.A$ are guarded recursive types that unfold to $A[\textcircled{\exists}(\mathit{Fix}\,\alpha.A)/\alpha]$ via the terms *into* and *out*. The most important of these types is $\mathit{Sig}\, A$ defined as $\mathit{Fix}\,\alpha.(A \times \alpha)$, which unfolds to $A \times \textcircled{\exists}(\mathit{Sig}\, A)$. That is, a signal consists of a current value and a delayed tail, which at some time in the future may return a new signal.

As an example, we can use any push channel $\kappa :_c A \in \Delta$, i.e., $c \in \{p, bp\}$, to define a corresponding delayed signal $\mathit{sigAwait}_{\kappa} : \textcircled{\exists}(\mathit{Sig}\, A)$ in *Async RaTT*:

$$\mathit{sigAwait}_{\kappa} = \mathit{fix}\, x.\mathit{delay}_{\mathit{cl}(\mathit{wait}_{\kappa})}\ (\mathit{into}\ (\mathit{adv}\, \mathit{wait}_{\kappa}, \mathit{adv}_{\forall}\, x))$$

where the recursion variable x has type $\textcircled{\forall}(\textcircled{\exists}(\mathit{Sig}\, A))$. Since $\mathit{sigAwait}_{\kappa}$ is typeable in an empty typing context, we can also form the stable signal $\mathit{box}\, \mathit{sigAwait}_{\kappa} : \square(\textcircled{\exists}(\mathit{Sig}\, A))$. The signal $\mathit{sigAwait}_{\kappa}$ operates 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.

Aside from all of 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.

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. In addition, we elide the clock subscript

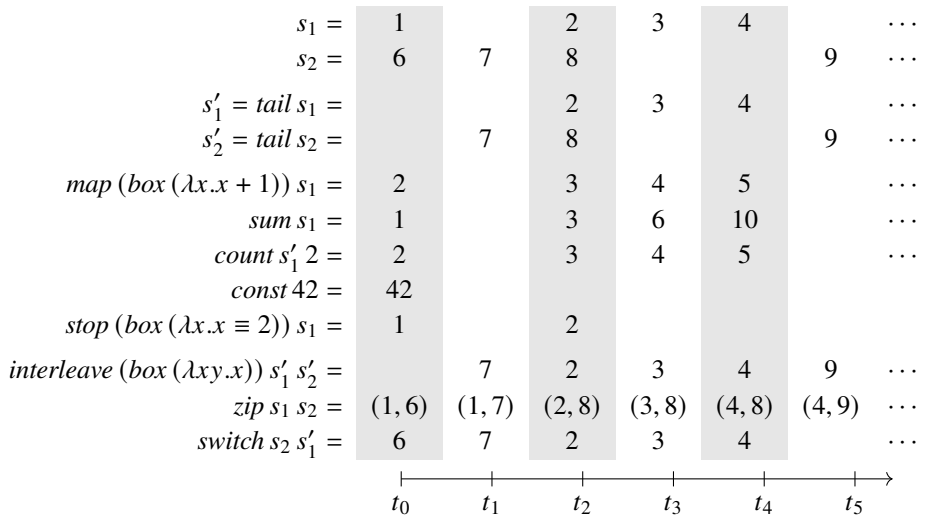


Fig. 4. Example (delayed) signals produced by signal combinators.

of $delay_\theta$ when it can be inferred from the context, which is the case in most instances. This syntax can be easily elaborated into the *Async RaTT* calculus as described in section 3.6.

We use a syntax that closely resembles Haskell. As a consequence, the examples we present in this section can be easily adapted to *Async Rattus*, an implementation of *Async RaTT* as a Haskell library (Bahr et al. 2024a). For more examples of practical programs, Async Rattus also provides a GUI library (Disch et al. 2025), a property-based testing library (Nielsen et al. 2026), and an implementation of Elliott's *Push-Pull FRP* (Elliott 2009; Faurby Klausen et al. 2026).

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 $Sig\ A$ defined as $Fix\,\alpha.(A \times \alpha)$. For readability, we use the shorthand $s :: t$ for $into\ (s, t)$, such that, given $s : A$ and $t : \textcircled{\exists}(Sig\ A)$, we have that $s :: t : Sig\ A$. Since this shorthand notation only uses introduction forms, we can also use it for pattern matching (cf. section 3.6). For example, we can extract the head and the tail of a signal as follows:

$$head : Sig\ A \to A$$
$$head\ (x :: xs) = x$$

$$tail : Sig\ A \to \textcircled{\exists}\ (Sig\ A)$$
$$tail\ (x :: xs) = xs$$

Figure 4 illustrates the signals produced by some of the signal combinators we present in this section. At the top, the figure shows two signals, s_1 and s_2, which produce new values independently of each other. Both have an initial value at time t_0, but at the next time instant t_1 only s_2 produces a new value. That is, the clock of s_2's tail ticks at t_1, but the clock of s_1's tail ticks at t_2. Underneath s_1 and s_2, we find signals derived from s_1 and s_2 using the signal combinators that we implement in the following.

We start with the implementation of one of the simplest signal combinators:

$$map : \Box\ (A \to B) \to Sig\ A \to Sig\ B$$
$$map\ f\ (x :: xs) = unbox\ f\ x :: delay\ (map\ f\ (adv\ xs))$$

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. The need for the $\Box$ modality also has an intuitive justification: The function f must be applied to all future values of the input signal, 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:

$$scan : stable\ B \Rrightarrow \Box\ (B \to A \to B) \to B \to Sig\ A \to Sig\ B$$
$$scan\ f\ acc\ (a :: as) = acc' :: delay\ (scan\ f\ acc'\ (adv\ as))$$
where $acc' = unbox\ f\ acc\ a$

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 $\Rrightarrow$ 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:

$$sum : Sig\ Nat \to Sig\ Nat$$
$$sum = scan\ (box\ (\lambda\, m\ n.\, m + n))\ 0$$

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

$$sigAwait_\kappa : \Diamond\!\!\!\!-\ (Sig\ A)$$
$$sigAwait_\kappa = delay\ (adv\ wait_\kappa :: sigAwait_\kappa)$$

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

$$scanAwait : stable\ B \Rrightarrow \Box\ (B \to A \to B) \to B \to \Diamond\!\!\!\!-\ (Sig\ A) \to Sig\ B$$
$$scanAwait\ f\ acc\ as = acc :: delay\ (scan\ f\ acc\ (adv\ as))$$

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:

$$count : \Diamond\!\!\!\!-\ (Sig\ A) \to Nat \to Sig\ Nat$$
$$count\ s\ n = scanAwait\ (box\ (\lambda\, m\ \_.\, m + 1))\ n\ s$$

The argument n is the initial value of the counter. That is, the signal $count\ s\ n$ produces the numbers $n, n + 1, n + 2$, etc., updating each time s produces a new value.

Like *scan*, also *map* naturally has a variant for delayed signals:

$$mapAwait : \Box\ (A \to B) \to \Diamond\!\!\!-\ (Sig\ A) \to \Diamond\!\!\!-\ (Sig\ B)$$
$$mapAwait\ f\ d = delay\ (map\ f\ (adv\ d))$$

The advantage of signals being first-class elements of the language is that we can easily combine simple signals to construct more complex signals. The simplest signal that can serve as such a basic building block for more complex signals is the constant signal:

$$const : A \to Sig\ A$$
$$const\ x = x :: never$$

In isolation this combinator may appear to be of little use. Its utility becomes apparent once we also have means to combine it with other signals. A very simple way to combine signals is provided by the following *jump* combinator:

$$jump : \Box\ (A \to (1 + Sig\ A)) \to Sig\ A \to Sig\ A$$
$$\begin{aligned} jump\ f\ (x :: xs) = \mathbf{case}\ unbox\ f\ x\ \mathbf{of} \\ \quad in_1\ ()\ .x :: delay\ (jump\ f\ (adv\ xs)) \\ \quad in_2\ xs'.xs' \end{aligned}$$

This combinator produces a signal that first behaves like the second argument, but as soon as the first argument produces a new signal when given the current value of the signal, it behaves like this new signal.

We can now use *const* to define a combinator that takes a predicate and a signal and modifies the signal so that it stops, i.e., behaves like the constant signal, as soon as the predicate is satisfied:

$$stop : \Box\ (A \to Bool) \to Sig\ A \to Sig\ A$$
$$stop\ p = jump\ (box\ (\lambda x.\ \mathbf{if}\ unbox\ p\ x\ \mathbf{then}\ in_2\ (const\ x)\ \mathbf{else}\ in_1\ ()))$$

For readability, we have used the notation *Bool* as shorthand for $1 + 1$ and **if** b **then** d **else** e as shorthand for $case\ b\ of\ in_1\ x.d; in_2\ y.e$.

We can use *stop* to implement a counter with an upper bound:

$$countMax : \Diamond\!\!\!-\ (Sig\ A) \to Nat \to Nat \to Sig\ Nat$$
$$countMax\ s\ start\ end = stop\ (box\ (\lambda n.\ n \equiv end))\ (count\ s\ start)$$

For the above example, we assume that we have implemented a comparison operator $\equiv$ on natural numbers, which we can do using rec_{Nat}.

It is natural to ask whether we can implement a *filter* combinator that takes a signal $s : Sig\ A$ along with a predicate $p : \Box(A \to Bool)$ and produces a new signal that behaves like s but restricted to only those values that satisfy p. However, if we try to implement such a combinator, we quickly get stuck:

$$filter : \Box\ (A \to Bool) \to Sig\ A \to Sig\ A$$
$$filter\ p\ (x :: xs) = (\mathbf{if}\ unbox\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ ???) :: delay\ (filter\ p\ (adv\ xs))$$

At the position marked with question marks, we have to come up with a value of type A. But the only value we have at hand, namely $x : A$, does not satisfy the predicate p.

The inability to implement a filter combinator of the above type is a direct consequence of *Async RaTT*'s productivity guarantee: If we could implement *filter*, we could construct a non-productive signal $filter\ (box(\lambda x.false))\ s$ from any signal s. However, the productivity guarantee is not an essential hindrance to filtering, as we might still hope to implement

filtering of delayed signals, which would not contradict productivity. Alas, *Async RaTT* cannot accommodate such a variant of *filter* either, and we get stuck again:

$$\begin{aligned}
&\mathit{filterAwait} : \Box\ (A \to \mathit{Bool}) \to \Diamond\!\!\!-\ (\mathit{Sig}\ A) \to \Diamond\!\!\!-\ (\mathit{Sig}\ A)\\
&\mathit{filterAwait}\ p\ \mathit{xs} = \mathit{delay}\ (\mathbf{let}\ (x :: \mathit{xs}') = \mathit{adv}\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ (\mathbf{if}\ \mathit{unbox}\ p\ x\ \mathbf{then}\ x\ \mathbf{else}\ ???) :: \mathit{filterAwait}\ p\ \mathit{xs}')
\end{aligned}$$

The issue is that we can only consume a delayed computation (such as *xs* above) and then check whether the value it produces satisfies a predicate if we also promise to construct a new delayed computation. This can be seen in the above failed implementation of *filterAwait*: When we advance *xs* using *adv*, we can only do so in the context of a surrounding *delay*, which thus forces us to construct a suitable signal of type $\mathit{Sig}\ A$. Put more succinctly, we cannot skip a $\Diamond\!\!\!-$ modality when traversing a signal. In particular, there is no general operation of type $\Diamond\!\!\!-\ (\Diamond\!\!\!-\ A) \to \Diamond\!\!\!-\ A$.

3.2 Concurrent signal combinators

The combinators we have looked at so far only consume a single signal, and thus have 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:

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

In this and subsequent definitions, we use the shorthands *Left*, *Right*, and *Both* to construct and pattern match values of type $((A_1 \times \Diamond\!\!\!-A_2) + (\Diamond\!\!\!-A_1 \times A_2)) + (A_1 \times A_2)$ produced by *select*. For example, *Left s t* is short for $\mathit{in}_1(\mathit{in}_1(s,t))$, 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 the 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 could use *zip*:

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

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 take the previously observed value from the signal that did not tick and copy it into the future. Hence, we need both types,

A and B, to be stable: The variables x of type A and y of type B are bound outside the scope of the *delay*, but they are used inside it, namely in the *Right* case and the *Left* case, respectively.

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:

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

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 two variants of *switch*, both of which allow the new signal to depend on the value of the previous signal:

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

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.

We can further generalise *switchS* so that instead of waiting for a single delayed function to produce a new signal, we wait for a delayed *signal* of such functions:

$$\begin{aligned}
&\mathit{switchR} : \mathit{stable}\ A \Rightarrow \mathit{Sig}\ A \to \Diamond\ (\mathit{Sig}\ (A \to \mathit{Sig}\ A)) \to \mathit{Sig}\ A\\
&\mathit{switchR}\ \mathit{sig}\ \mathit{steps} = \mathit{switchS}\ \mathit{sig}\\
&\quad (\mathit{delay}\ (\mathbf{let}\ \mathit{step} :: \mathit{steps}' = \mathit{adv}\ \mathit{steps}\ \mathbf{in}\ \lambda x.\ \mathit{switchR}\ (\mathit{step}\ x)\ \mathit{steps}'))
\end{aligned}$$

While *switchS* changes behaviour *once*, namely when the delayed function arrives, *switchR* allows us to change behaviour repeatedly, namely every time the delayed signal produces a new function.

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 *Nat* and are thus described by signals of type *Sig Nat*. The number displayed in the text fields should be incremented each time the user clicks a button, which is available as an input channel $up :_p 1 \in \Delta$. However, there is only one 'up' button and the user can change which

text field should be changed by the 'up' button with the help of a 'toggle' button, which is available as an input channel $toggle :_p 1 \in \Delta$.

That means, the contents of the first text field can be described by a signal produced by the *count* combinator, but then switches to a signal produced by the *const* combinator as soon as '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:

$$\begin{aligned}
&toggleSig : stable\ A \Rightarrow \Box\ (\Diamond\!\!\!-\ 1) \to \Box\ (A \to Sig\ A) \to \Box\ (A \to Sig\ A) \to A \to Sig\ A \\
&toggleSig\ tog\ f\ g\ x = switchS\ (unbox\ f\ x)\ (delay_{cl(tick)}\ (toggleSig\ tog\ g\ f)) \\
&\quad \textbf{where}\ tick = unbox\ tog
\end{aligned}$$

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. We use *switchS* to construct a signal that starts like the first signal provided by f, but then switches to g as soon as the toggle *tog* ticks via a recursive call that swaps the order of the two arguments f and g.

Note that *toggleSig* is one of the few examples that requires *delay* to have an explicit clock annotation as it cannot be inferred from the context. Such explicit annotations are necessary when the argument of *delay* does not contain a relevant occurrence of *adv* or *select*, as is the case for *toggleSig*. Further note that we bind the term *unbox tog* to the variable *tick* before using it to construct the clock $cl(tick)$. This indirection is necessary in the type system of *Async RaTT*, which only allows clocks to be constructed from values. We will lift this restriction later in section 6 when we introduce *Full Async RaTT*.

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

$$\begin{aligned}
&field1, field2 : Sig\ Nat \\
&field1 = toggleSig\ (box\ wait_{toggle})\ (box\ (count\ sigAwait_{up}))\ (box\ const) \quad 0 \\
&field2 = toggleSig\ (box\ wait_{toggle})\ (box\ const)\ (box\ (count\ sigAwait_{up}))\ 0
\end{aligned}$$

Note that the dataflow graph changes during the execution of the program and how that change is reflected in the clocks associated with the output channels: The output channel for the first text field first has the clock $\{up, toggle\}$ 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 the output channel for the first text field changes to $\{toggle\}$ as it now ignores the 'up' button. We will examine the runtime behaviour of this example in more detail in section 4.3 when we discuss the operational semantics of *Async RaTT*.

3.4 An interactive timer

As a second – more complex – GUI example, we consider an interactive timer based on the 7GUIs benchmark (Kiss 2014). This timer is a natural number signal that, starting from 0, counts upwards each second. This signal depends on three push-only channels: $seconds :_p 1$, $reset :_p 1$, and $max :_p Nat$. The timer is incremented by 1 each time *seconds* ticks and is reset to 0 whenever *reset* ticks. Moreover, the timer stops when it reaches a maximum value, which is provided by the *max* channel.

Our goal is to implement a signal of type $Sig\ Nat$ that has the behaviour described above. To this end, we implement a signal of type $Sig\ (Nat \times Nat)$ that consists of the actual timer value as well as the current maximum of the timer. With this in mind, we can implement the function that starts the basic behaviour of the timer given an initial value:

$$
\begin{aligned}
&start : (Nat \times Nat) \to Sig\ (Nat \times Nat)\\
&start\ (n, max) = stop\\
&\quad (box\ (\lambda\,(n, max).\, n \geqslant max))\\
&\quad (scanAwait\ (box\ (\lambda\,(n, max)\ \_.\ (n + 1, max)))\ (n, max)\ sigAwait_{seconds})
\end{aligned}
$$

We use *scanAwait* to increment the timer similarly to the *count* example from section 3.1. Then we use the *stop* combinator to stop the signal once we have reached the maximum.

Next we define two signals that describe how the *reset* and *max* channels change the timer:

$$
\begin{aligned}
&resetSig : \Diamond\!\!\!-\ (Sig\ (Nat \times Nat \to Nat \times Nat))\\
&resetSig = mapAwait\ (box\ (\lambda\,()\ (\_, max).\, (0, max)))\ sigAwait_{reset}
\end{aligned}
$$

$$
\begin{aligned}
&setMaxSig : \Diamond\!\!\!-\ (Sig\ (Nat \times Nat \to Nat \times Nat))\\
&setMaxSig = mapAwait\ (box\ (\lambda\, max'\ (n, \_).\, (min\ n\ max', max')))\ sigAwait_{max}
\end{aligned}
$$

The two delayed signals *resetSig* and *setMaxSig* produce functions that describe how the current state of the timer should be changed upon receiving input on the *reset* and *max* channels, respectively. To this end, we assume that $min : Nat \to Nat \to Nat$ implements a function that produces the minimum of two natural numbers. We then interleave these two signals via the *interleave* combinator together with function composition as the tiebreaker, i.e., if both signals produce a function at the same time we compose them sequentially:

$$
\begin{aligned}
&inputSig : \Diamond\!\!\!-\ (Sig\ (Nat \times Nat \to Nat \times Nat))\\
&inputSig = interleave\ (box\ (\circ))\ resetSig\ setMaxSig
\end{aligned}
$$

We then compose the functions produced by *inputSig* with the *start* function to produce functions that describe how to restart the timer signal in reaction to a *reset* or *max* input. We can then use this as the switching signal for the *switchR* combinator to obtain the desired behaviour:

$$
\begin{aligned}
&restartSig : \Diamond\!\!\!-\ (Sig\ (Nat \times Nat \to Sig\ (Nat \times Nat)))\\
&restartSig = mapAwait\ (box\ (\lambda f.\, start\ \circ\ f))\ inputSig
\end{aligned}
$$

$$
\begin{aligned}
&timerMaxSig : Sig\ (Nat \times Nat)\\
&timerMaxSig = switchR\ (start\ (0, 100))\ restartSig
\end{aligned}
$$

$$
\begin{aligned}
&timerSig : Sig\ Nat\\
&timerSig = map\ (box\ (\lambda\,(n, \_).\, n))\ timerMaxSig
\end{aligned}
$$

The final signal *timerSig* is then produced by projecting to the first component, i.e., forgetting the component that stores the maximum.

3.5 Integral and derivative

Buffered-push 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 $\kappa :_{bp} A \in \Delta$, we can construct the following signal (using $sigAwait_{\kappa}$ from section 3.1):

```
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. 5. Integral and derivative signal combinators.

```
sig_κ : Sig A
sig_κ = read_κ :: sigAwait_κ
```

To illustrate what we can do with such input signals, we assume that *Async RaTT* is extended with a stable type *Float* together with typical operations on floating-point numbers. Figure 5 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 $sample :_{bp} Float \in \Delta$ 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 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 will not change until the underlying signal has a non-zero value. Hence, we do not 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 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 that happens, the signal behaves as if *sample* has just ticked in order to provide a timely update of the derivative.

These two combinators can be easily generalised from floating-point values to any vector space. Indeed, the standard library of Async Rattus (Bahr et al. 2024b) implements *integral* and *derivative* in this generalised fashion. In combination with the other signal combinators such as *switch* and *scan*, these generalised *integral* and *derivative* combinators can be used to describe complex behaviours on multidimensional data.

3.6 Elaboration of surface syntax into core calculus

To illustrate how the surface language used in the examples above elaborates into the *Async RaTT* core calculus, we reconsider the definition of *map*:

$$map : \Box\ (A \to B) \to Sig\ A \to Sig\ B$$
$$map\ f\ (x :: xs) = unbox\ f\ x :: delay\ (map\ f\ (adv\ xs))$$

The syntactic sugar of this definition elaborates to the following term in plain *Async RaTT*:

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

Recall that $s :: t$ is a shorthand for $into\,(s,t)$. 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 $fix\, r.t$ so that the recursive occurrence of *map* is replaced by $adv_\forall\ r$. Hence, recursive calls must always occur in the scope of a $\checkmark$, which is the case in the definition of *map*, whose recursive call appears in the scope of a *delay*. The syntactic sugar elides the subscript $cl\,(xs)$ of *delay*, which can be uniquely inferred from the fact that we have the term $adv\ xs$ 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 $\checkmark$. We can think of these top-level definitions as being implicitly boxed when defined and unboxed when used later on. That is, these definitions are translated as follows to the core calculus:

$$let\ scan = box(\ldots)\ in$$
$$let\ scanAwait = box(\ldots(unbox\ scan)\ldots)\ in$$
$$\ldots$$

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 achieve 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

The evaluation semantics is given 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 has 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.

Up to this point we have used the term *clock* to refer to both clock expressions (i.e., expressions involving $cl(\cdot)$ and clock union $\sqcup$) and clocks (i.e., sets of push channels). When discussing the operational semantics, we have to be more precise about the distinction between these two concepts. To this end, we use θ to refer to a clock expression as defined in Figure 2, and we use Θ to refer to a clock, which is defined as a finite set of push channels. When the machine encounters a closed clock expression θ, it has to evaluate θ to a corresponding clock $|\theta|$:

$$|cl(l)| = cl(l) \qquad |cl(wait_{\kappa})| = \{\kappa\} \qquad |\theta \sqcup \theta'| = |\theta| \cup |\theta'|$$

This evaluation of clock expressions occurs, for example, in the semantics for $delay_{\theta}$, which we return to in more detail shortly.

To facilitate the delay of computations and their resumption at a later time, the syntax of the language features heap locations l, which are not typeable 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 $cl(l)$. As soon as the clock $cl(l)$ 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 $cl(l) = \Theta$. A clock Θ is a finite set of push channels drawn from Δ, and it ticks whenever any of its channels $\kappa \in \Theta$ is updated. For example, assuming an input context Δ for a GUI, the clock $\{keyPressed, mouseCoord\}$ ticks whenever the user presses a key or moves the mouse.

$$\frac{}{\langle v;\sigma\rangle \Downarrow^{\iota} \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow^{\iota} \langle v;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow^{\iota} \langle v';\sigma''\rangle}{\langle (t,t');\sigma\rangle \Downarrow^{\iota} \langle (v,v');\sigma''\rangle}$$

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

$$\frac{\langle t;\sigma\rangle \Downarrow^{\iota} \langle in_i(v);\sigma'\rangle \qquad \langle t_i[v/x];\sigma'\rangle \Downarrow^{\iota} \langle v_i;\sigma''\rangle \qquad i\in\{1,2\}}{\langle case\, t\, of\, in_1\, x.t_1; in_2\, x.t_2;\sigma\rangle \Downarrow^{\iota} \langle v_i;\sigma''\rangle}$$

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

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

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

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

$$\frac{\kappa\in |cl(v_i)|\setminus|cl(v_{3-i})| \qquad \langle adv\, v_i;\eta_N\langle\kappa\mapsto w\rangle\eta_L\rangle \Downarrow^{\iota} \langle u_i;\sigma\rangle \qquad u_{3-i} = v_{3-i} \qquad i\in\{1,2\}}{\langle select\, v_1\, v_2;\eta_N\langle\kappa\mapsto w\rangle\eta_L\rangle \Downarrow^{\iota} \langle in_1(in_i\,(u_1,u_2));\sigma\rangle}$$

$$\frac{\kappa\in |cl(v_1)|\cap|cl(v_2)| \qquad \langle adv\, v_1;\eta_N\langle\kappa\mapsto w\rangle\eta_L\rangle \Downarrow^{\iota} \langle u_1;\sigma\rangle \qquad \langle adv\, v_2;\sigma\rangle \Downarrow^{\iota} \langle u_2;\sigma'\rangle}{\langle select\, v_1\, v_2;\eta_N\langle\kappa\mapsto w\rangle\eta_L\rangle \Downarrow^{\iota} \langle in_2\,(u_1,u_2)\,;\sigma'\rangle}$$

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

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

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

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

$$\frac{\langle t;\sigma\rangle \Downarrow^{\iota} \langle into\, v;\sigma'\rangle}{\langle out\, t;\sigma\rangle \Downarrow^{\iota} \langle v;\sigma'\rangle}$$

Fig. 6. Evaluation semantics.

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 $dom\,(\eta)$, each have a clock that contains a given input channel κ:

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

It is safe to evaluate terms stored in a heap $\eta \in Heap^{\kappa}$ 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 $\eta_N \in Heap^{\kappa}$ and $dom\,(\eta_N) \cap dom\,(\eta_L) = \emptyset$. 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, which stores terms that are safe to be evaluated now. The $\langle\kappa \mapsto v\rangle$ component of a two-heap store, also referred to as a tick, 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. We write $dom\,(\sigma)$ for the combined set of locations in a store σ, i.e., $dom\,(\eta_N\,\langle\kappa \mapsto v\rangle\,\eta_L) = dom\,(\eta_N) \cup dom\,(\eta_L)$.

The complete definition of the evaluation semantics is given in Figure 6. The lambda calculus fragment with product and sum types follows the standard call-by-value semantics. We review the remaining aspects that are specific to *Async RaTT* below.

Let us first consider the semantics for *delay*: To allocate fresh locations in the store, we assume a function *alloc*, which, if given a clock Θ and a store σ, produces a location $l \notin dom\,(\sigma)$ with $cl(l) = \Theta$. The machine then stores the argument term t at that newly allocated location l, which 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 $l \mapsto t$.

Also *never* allocates a fresh heap location l, but it does not store any term at that location. Since the allocated location l has the empty clock – which never ticks – the machine will never try to read from location l.

The semantics for *adv* has two cases: In the case for $adv\,wait_{\kappa}$, the machine simply looks up the value v that we have received on channel κ. In the case $adv\,l$, the semantics retrieves a previously delayed computation. The typing discipline ensures that $adv\,v$ will only be evaluated in the context of a store of the form $\eta_N\,\langle\kappa \mapsto w\rangle\,\eta_L$, where either $v = wait_{\kappa}$ with a matching channel κ, or v is a location $l \in dom\,(\eta_N)$ and therefore also $\kappa \in cl(l)$.

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 of the form $wait_{\kappa'}$, 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 typeable in the calculus but is introduced by the machine. Intuitively speaking, we can think of a value of the form $dfix\,x.t$ as shorthand for $\Lambda\theta.(delay_{\theta}(fix\,x.t))$. That is, $dfix\,x.t$ 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 $adv_{\triangledown}$ combinator for $\triangledown$ can thus also be interpreted as first providing the clock θ and then advancing the delayed computation $delay_{\theta}(fix\,x.t)$, which means evaluating $fix\,x.t$.

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

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

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

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

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

Fig. 7. Reactive semantics.

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}, which only contains 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 $\Delta \Rightarrow \Gamma_{out}$.

Given an output context $\Gamma_{out} = x_1 : A_1, \ldots, x_n : A_n$, we define the type $\mathit{Sigs}\,(\Gamma_{out})$ as the product of signal types constructed from the types in Γ_{out}, i.e., $\mathit{Sigs}\,(\Gamma_{out}) = \mathit{Sig}\, A_1 \times \cdots \times \mathit{Sig}\, A_n$. This n-ary product type is 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 $\Delta \Rightarrow \Gamma_{out}$, denoted $t : \Delta \Rightarrow \Gamma_{out}$, if $\vdash_{\Delta} t : \mathit{Sigs}\,(\Gamma_{out})$.

The operational semantics of a reactive program is described by the machine in Figure 7. 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 $t : \Delta \Rightarrow \Gamma_{out}$ 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 triple $\langle D; \sigma; \iota\rangle$, where ι is the current input buffer and D is a sequence of the form $x_1 \mapsto l_1, \ldots, x_n \mapsto l_n$ that maps each output channel $x_i \in \mathit{dom}\,(\Gamma_{out})$ to a heap location. That is, D 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:

$$\begin{array}{ll} \text{an initialisation transition} & \langle t;\iota\rangle \overset{O}{\Longrightarrow} \langle D;\eta;\iota\rangle \\ \text{an input transition} & \langle D;\eta;\iota\rangle \overset{\kappa\mapsto v}{\Longrightarrow} \langle D;\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L;\iota'\rangle \\ \text{an output transition} & \langle D;\eta_N\,\langle\kappa\mapsto v\rangle\,\eta_L;\iota\rangle \overset{O}{\Longrightarrow} \langle D';\eta;\iota\rangle \end{array}$$

where O is a sequence $y_1 \mapsto v_1, \ldots, y_m \mapsto v_m$ that maps some output channels $y_1 : B_1, \ldots, y_m : B_m \in \Gamma_{out}$ to values. After the initialisation 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 $(v_1 :: l_1, \ldots, v_m :: l_m)$, where each component $v_i :: l_i$ corresponds to an output channel $x_i : A_i \in \Gamma_{out}$. 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 where clocks contain κ and a part where clocks do not:

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

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

Finally, the output transition checks for each element $x \mapsto l$ in D 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 *Async RaTT* term without syntactic sugar as described in section 3.6:

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

During the execution, the machine turns fixed points like *toggleSig* into delayed fixed points that use *dfix* instead of *fix*. We write $\mathit{toggleSig}'$ for this delayed fixed point, i.e., $\mathit{toggleSig}'$ is obtained from *toggleSig* by replacing *fix* with *dfix*. We will use the same notational

convention for other fixed point definitions and write $sigAwait'_\kappa$ and $scan'$ for the *dfix* versions of $sigAwait_\kappa$ and *scan* from section 3.1.

For the sake of simplicity, we consider a reactive program with only one output channel, namely the program $field1 : \Delta \Rightarrow \Gamma_{out}$ with

$$\begin{aligned} field1 &= toggleSig\, t\, s_1\, s_2\, 0 & \Delta &= \{up :_p 1, toggle :_p 1\} \\ t &= box\, wait_{toggle} & \Gamma_{out} &= x : Nat \\ s_1 &= box\, (count\, sigAwait_{up}) \\ s_2 &= box\, const \end{aligned}$$

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:

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

$$\begin{aligned} \text{where}\quad & l_1 \mapsto case\, select\, l_3\, l_4\, of\, \ldots & & cl(l_1) = \{toggle, up\} \\ & l_2 \mapsto adv\, wait_{up} :: adv_\forall\, sigAwait'_{up} & & cl(l_2) = \{up\} \\ & l_3 \mapsto scan\, (box\, (\lambda m.\lambda n.m + 1))\, 0\, (adv\, l_2) & & cl(l_3) = \{up\} \\ & l_4 \mapsto adv_\forall\, toggleSig'\, t\, s_2\, s_1 & & cl(l_4) = \{toggle\} \end{aligned}$$

We can see that the next value for the output channel x is provided by the delayed computation at location l_1, and since $cl(l_1) = \{toggle, up\}$, 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, the machine produces the following two transitions:

$$\begin{aligned} \langle x \mapsto l_1; l_1, l_2, l_3, l_4; \emptyset\rangle &\overset{up\mapsto ()}{\Longrightarrow} \langle x \mapsto l_1; l_1, l_2, l_3\, \langle up \mapsto ()\rangle\, l_4; \emptyset\rangle \\ &\overset{x\mapsto 1}{\Longrightarrow} \langle x \mapsto l_5; l_4, l_5, l_6, l_7; \emptyset\rangle \end{aligned}$$

$$\begin{aligned} \text{where}\quad & l_5 \mapsto case\, select\, l_7\, l_4\, of\, \ldots & & cl(l_5) = \{toggle, up\} \\ & l_6 \mapsto adv\, wait_{up} :: adv_\forall\, sigAwait'_{up} & & cl(l_6) = \{up\} \\ & l_7 \mapsto adv_\forall\, scan'\, (box\, (\lambda m.\lambda n.m + 1))\, 1\, (adv\, l_6) & & cl(l_7) = \{up\} \end{aligned}$$

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

$$\begin{aligned} \langle x \mapsto l_5; l_4, l_5, l_6, l_7; \emptyset\rangle &\overset{toggle\mapsto ()}{\Longrightarrow} \langle x \mapsto l_5; l_4, l_5\, \langle toggle \mapsto ()\rangle\, l_6, l_7; \emptyset\rangle \\ &\overset{x\mapsto 1}{\Longrightarrow} \langle x \mapsto l_8; l_6, l_7, l_8, l_9; \emptyset\rangle \end{aligned}$$

$$\begin{aligned} \text{where}\quad cl(l_0) = \emptyset \quad & l_8 \mapsto case\, select\, l_0\, l_9\, of\, \ldots & & cl(l_8) = \{toggle\} \\ & l_9 \mapsto adv_\forall\, toggleSig'\, t\, s_1\, s_2 & & cl(l_9) = \{toggle\} \end{aligned}$$

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:

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

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, causality, the absence of implicit space leaks, and signal independence. We address each of them in turn.

4.4.1 Productivity

Reactive programs $t : \Delta \Rightarrow \Gamma_{out}$ 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 $\kappa :_b A \in \Delta$ or $\kappa :_{bp} A \in \Delta$.
- 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 sequence of output values O is well-typed, written $\vdash O : \Gamma_{out}$, if for all $x \mapsto v \in O$, we have that $x : A \in \Gamma_{out}$ and $\vdash v : A$.

We can now formally state the productivity property as follows:

Theorem 4.1 (productivity)**.** *Given a reactive program $t : \Delta \Rightarrow \Gamma_{out}$, 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*

$$\langle t; \iota_0\rangle \overset{O_0}{\Longrightarrow} \langle D_0; \eta_0; \iota_0\rangle \overset{\kappa_0\mapsto v_0}{\Longrightarrow} \langle D_0; \sigma_0; \iota_1\rangle \overset{O_1}{\Longrightarrow} \langle D_1; \eta_1; \iota_1\rangle \overset{\kappa_1\mapsto v_1}{\Longrightarrow} \dots$$

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

Note that in the above transition sequence, the machine states alternate between stores consisting of a single heap η_i and stores σ_i, which are of the form $\eta_{N,i}\,\langle \kappa_i \mapsto v_i\rangle\,\eta_{L,i}$.

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

4.4.2 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

$$\langle t; \iota_0\rangle \overset{O_0}{\Longrightarrow} \langle D_0; \eta_0; \iota_0\rangle \overset{\kappa_0\mapsto v_0}{\Longrightarrow} \langle D_0; \sigma_0; \iota_1\rangle \overset{O_1}{\Longrightarrow} \langle D_1; \eta_1; \iota_1\rangle \overset{\kappa_1\mapsto v_1}{\Longrightarrow} \dots \tag{2}$$

each sequence 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 that* (2) *and the following transition sequence are well-typed:*

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

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

4.4.3 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

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

all heap locations l in η that depend on κ, i.e., those with $\kappa \in cl(l)$, are moved into η_N after the input transition, and then η_N is garbage collected after the output transition, so that the resulting heap η' only consists of η_L together with the delayed computations allocated during the output transition. That is, a delayed computation at location l is only kept in memory until its clock $cl(l)$ ticks. Likewise, the input $\kappa \mapsto v$ is garbage collected after each output transition. The machine only keeps the latest value of buffered channels in its buffer ι. The productivity property demonstrates that this aggressive garbage collection strategy is safe: The machine never gets stuck due to an attempt to dereference a garbage-collected heap location.

4.4.4 Signal independence

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

First, values received on buffered-only channels will never produce an output:

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

$$\langle t; \iota_0\rangle \overset{O_0}{\Longrightarrow} \langle D_0; \eta_0; \iota_0\rangle \overset{\kappa_0\mapsto v_0}{\Longrightarrow} \langle D_0; \sigma_0; \iota_1\rangle \overset{O_1}{\Longrightarrow} \langle D_1; \eta_1; \iota_1\rangle \overset{\kappa_1\mapsto v_1}{\Longrightarrow} \dots$$

a well-typed transition sequence for t. *If* $\kappa_i :_b A \in \Delta$ *for some* A, *then* O_{i+1} *is empty,* $\sigma_i = \emptyset \langle \kappa_i \mapsto v_i\rangle \eta_i$, $\eta_{i+1} = \eta_i$, *and* $D_{i+1} = D_i$.

As a practical consequence, buffered-only channels can be used to model continuous inputs of a reactive system, such as the current time or sensor measurements that can be sampled at any point in time. More concretely, to implement such continuous inputs, the runtime system for an implementation of *Async RaTT* can react to an update $\kappa \mapsto v$ on a push channel κ, by first performing an input transition $\overset{\kappa_i \mapsto v_i}{\Longrightarrow}$ for each corresponding buffered-only channel κ_i before finally performing the input transition $\overset{\kappa \mapsto v}{\Longrightarrow}$ to then obtain the output O:

$$\begin{aligned}
\langle D; \eta; \iota_0 \rangle &\overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle D; \emptyset \langle \kappa_0 \mapsto v_0 \rangle \eta; \iota_1 \rangle \overset{\emptyset}{\Longrightarrow} \langle D; \eta; \iota_1 \rangle \\
&\overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \quad \dots \\
&\quad \vdots \\
&\overset{\kappa_n \mapsto v_n}{\Longrightarrow} \langle D; \emptyset \langle \kappa_n \mapsto v_n \rangle \eta; \iota_{n+1} \rangle \overset{\emptyset}{\Longrightarrow} \langle D; \eta; \iota_{n+1} \rangle \\
&\overset{\kappa \mapsto v}{\Longrightarrow} \langle D; \emptyset \langle \kappa \mapsto v \rangle \eta; \iota_{n+2} \rangle \overset{O}{\Longrightarrow} \langle D'; \eta'; \iota_{n+2} \rangle
\end{aligned}$$

By Proposition 4.4, the input from buffered-only channels leaves the sequence of delayed computations D and the heap η unchanged, and it does not produce any output.

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

Theorem 4.5 (push signal independence)**.** *Suppose* $(t_1, \dots, t_m) : \Delta \Rightarrow \Gamma_{out}$ *is a reactive program with* $\Gamma_{out} = x_1 : A_1, \dots, x_m : A_m$ *such that also* $t_j : \Delta' \Rightarrow (x_j : A_j)$ *is a reactive program for some* j *and* $\Delta' \subseteq \Delta$*. Given any well-typed transition sequence for* $(t_1, \dots, t_m)$

$$\langle (t_1, \dots, t_m) ; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle D_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle D_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle D_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots$$

then $x_j \mapsto v \in O_{i+1}$ *implies that* $\kappa_i \in dom\,(\Delta')$*. In other words, the output channel* x_j *is only updated when inputs in* Δ' *are updated.*

5 Metatheory

In this section, we prove 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* w and is defined by induction on the structure of the type A and a suitable well-founded order on the 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 $Fix\,\alpha.A$ unfold to $A[\triangleleft(Fix\,\alpha.A)/\alpha]$, we can define $[\![Fix\,\alpha.A]\!](n+1, \sigma)$ in terms of $[\![A]\!](n+1, \sigma)$ and $[\![Fix\,\alpha.A]\!](n, \sigma')$, 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 evaluated 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:

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

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

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

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

That is, $\sigma \sqsubseteq \sigma'$ iff σ' is obtained from σ by storing additional terms. The following lemma shows how the order $\sqsubseteq$ indeed captures how the store evolves during evaluation:

Lemma 5.1. *If* $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v; \sigma' \rangle$*, then* $\sigma \sqsubseteq \sigma'$.

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

Finally, the fourth change that we have to account for is that the machine may receive an input value $\kappa \mapsto v$, which is captured by the following order $\sqsubseteq^{\Delta}_{\checkmark}$ on stores:

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

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

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

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

This preorder does not include garbage collection, as it is only sound in certain circumstances (cf. Lemma 5.6 below). 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 two 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 $\kappa \notin cl(l)$. We achieve this by making the logical relation $[\![\circledcirc A]\!](n, \sigma)$ satisfy the following clock independence property:

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

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

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

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

The full definition of the Kripke logical relation is given in Figure 8. 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 on 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.

$$|\alpha| = |\circledcirc A| = |\textcircled{\scriptsize V} A| = |1| = |Nat| = 1$$
$$|A \times B| = |A + B| = |A \to B| = 1 + |A| + |B|$$
$$|\Box A| = |Fix\, \alpha.A| = 1 + |A|$$

Note that in the definition for $\mathcal{V}_\Delta[\![\circledcirc A]\!](n+1, \sigma)$ in Figure 8, we use the shorthand $\sigma(l)$ for $\eta_L(l)$, where η_L is the later heap of σ.

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. Given an open term $\Gamma \vdash_\Delta t : A$, we aim to show that $t\gamma \in \mathcal{T}_\Delta[\![A]\!](n, \sigma)$ for all suitable substitutions γ, where we write $t\gamma$ for the term obtained from t after applying the substitution γ. To characterise such suitable substitutions γ, we need a corresponding logical relation for contexts, whose definition is given at the bottom of Figure 8. The interpretation of $\checkmark_\theta$ occurrences in a context is quite technical but is essentially determined by the interpretation of $\textcircled{\scriptsize E}$ due to the requirement of being left adjoint (Birkedal et al. 2020).

The definition of $C_\Delta[\![\Gamma]\!](n, \sigma)$ follows roughly the structure of the store σ: If Γ contains a tick $\checkmark_\theta$, then σ must contain a tick $\langle \kappa \mapsto v \rangle$ on a channel κ contained in θ. Otherwise, $C_\Delta[\![\Gamma]\!](n, \sigma)$ is empty. Consequently, since stores may contain at most one tick, $C_\Delta[\![\Gamma]\!](n, \sigma)$ is only non-empty for contexts Γ that contain at most one tick. This restricted semantic definition suffices for our proof of the fundamental property.

To gain an intuitive understanding of why this definition of $C_\Delta[\![\Gamma]\!](n, \sigma)$ is sufficient, we can use the following observation: Any closed term $\vdash_\Delta t : A$ could in principle be typed in a slightly modified type system that allows at most one tick in the context and which replaces the typing rule for *delay* with the following rule:

$$\frac{\Gamma^{\bigcirc}, \checkmark_\theta \vdash_\Delta t : A}{\Gamma \vdash_\Delta delay_\theta\, t : \circledcirc A}$$

The context $\Gamma^{\bigcirc}$, which we define below, is tick-free so that $\Gamma^{\bigcirc}, \checkmark_\theta$ has exactly one tick.

The definition of $\Gamma^{\bigcirc}$ is motivated by the idea that any variable occurring to the left of a tick $\checkmark_\theta$ can only be used in two ways: Either it can be used directly via the variable introduction rule if its type is stable, or it can be used via *adv* or *select* if its type is of the form $\circledcirc A$. To

$$
\begin{aligned}
\mathcal{V}_\Delta[\![1]\!]\,(w) &= \{()\}\,,\\
\mathcal{V}_\Delta[\![Nat]\!]\,(w) &= \{suc^n\,0 \mid n \in \mathbb{N}\}\,,\\
\mathcal{V}_\Delta[\![A \times B]\!]\,(w) &= \left\{(v_1, v_2) \,\middle|\, v_1 \in \mathcal{V}_\Delta[\![A]\!]\,(w) \wedge v_2 \in \mathcal{V}_\Delta[\![B]\!]\,(w)\right\},\\
\mathcal{V}_\Delta[\![A + B]\!]\,(w) &= \left\{in_1\,v \,\middle|\, v \in \mathcal{V}_\Delta[\![A]\!]\,(w)\right\} \cup \left\{in_2\,v \,\middle|\, v \in \mathcal{V}_\Delta[\![B]\!]\,(w)\right\}\\
\mathcal{V}_\Delta[\![A \to B]\!]\,(w) &= \left\{\lambda x.t \,\middle|\, \forall w' \gtrsim^\Delta w, v \in \mathcal{V}_\Delta[\![A]\!]\,(w')\,.t[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(w')\right\}\\
\mathcal{V}_\Delta[\![\Box A]\!]\,(n, \sigma) &= \left\{box\,t \,\middle|\, t \in \mathcal{T}_\Delta[\![A]\!]\,(n, \emptyset)\right\}\\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize V} A]\!]\,(0, \sigma) &= \{dfix\,x.t \mid dfix\,x.t \text{ a closed term}\}\\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize V} A]\!]\,(n+1, \sigma) &= \left\{dfix\,x.t \,\middle|\, t[dfix\,x.t/x] \in \mathcal{T}_\Delta[\![A]\!]\,(n, \emptyset)\right\}\\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize E} A]\!]\,(0, \sigma) &= Loc_\Delta \cup \{wait_\kappa \mid \kappa :_c A \in \Delta, c \in \{p, bp\}\}\\
\mathcal{V}_\Delta[\![\textcircled{\scriptsize E} A]\!]\,(n+1, \sigma) &= \left\{ l \in Loc_\Delta \,\middle|\, \begin{array}{l} \forall \kappa \in cl(l)\,, \vdash v : \Delta(\kappa).\\ \sigma(l) \in \mathcal{T}_\Delta[\![A]\!]\left(n, \left[\left[gc(\sigma)\right]_{\kappa \in \langle \kappa \mapsto v\rangle}[gc(\sigma)]_{\kappa \notin}\right]_{cl(l)}\right) \end{array} \right\}\\
&\quad \cup \{wait_\kappa \mid \kappa :_c A \in \Delta, c \in \{p, bp\}\}\\
\mathcal{V}_\Delta[\![Fix\,\alpha.A]\!]\,(w) &= \left\{into\,v \,\middle|\, v \in \mathcal{V}_\Delta[\![A[\textcircled{\scriptsize E}(Fix\,\alpha.A)/\alpha]]\!]\,(w)\right\}
\end{aligned}
$$

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

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

Using

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

Fig. 8. Logical relation.

capture this pattern, we define the notion of a *later type*, which is any type of the form $\textcircled{\scriptsize E}A$ and any stable type. Using this terminology, we define the context $\Gamma^{\bigcirc}$ as follows:

$$
\cdot^{\bigcirc} = \cdot \qquad (\Gamma, \checkmark_\theta)^{\bigcirc} = \Gamma^{\Box} \qquad (\Gamma, x : A)^{\bigcirc} = \begin{cases} \Gamma^{\bigcirc}, x : A & \text{if } A \text{ is a later type}\\ \Gamma^{\bigcirc} & \text{otherwise} \end{cases}
$$

That is, $\Gamma^{\bigcirc}$ removes all ticks from Γ and only keeps variables that either occur to the left of a tick in Γ and are of stable type, or that do not occur to the left of a tick in Γ and are of a later type. For example, if Γ is the context

$$
x_1 : 1 \to Nat, x_2 : \textcircled{\scriptsize E}Nat, x_3 : \textcircled{\scriptsize V}(Nat \to Nat), \checkmark_{cl(x_2)}, x_4 : \textcircled{\scriptsize E}Nat, x_5 : Nat, x_6 : 1 \to Nat
$$

then $\Gamma^{\bigcirc} = x_3 : \textcircled{\scriptsize V}(Nat \to Nat), x_4 : \textcircled{\scriptsize E}Nat, x_5 : Nat$.

The following lemma shows that $\Gamma^{\bigcirc}, \checkmark_\theta \vdash_\Delta t : A$ whenever $\Gamma, \checkmark_\theta \vdash_\Delta t : A$, which allows us to assume the former instead of the latter in the proof of the fundamental property for *delay*.

Lemma 5.2. *If* $\Gamma, \Gamma' \vdash_\Delta t : A$ *and* Γ' *not tick-free, then* $\Gamma^\bigcirc, \Gamma' \vdash_\Delta t : A$.

Proof We proceed by induction on the structure of $\Gamma, \Gamma' \vdash_\Delta t : A$.

- Let $t = x$. If $x : A \in \Gamma'$, then $\Gamma^\bigcirc, \Gamma' \vdash_\Delta x : A$ follows immediately. If $x : A \in \Gamma$, then A must be stable and thus $x : A \in \Gamma^\bigcirc$ as well. Hence, we also have $\Gamma^\bigcirc, \Gamma' \vdash_\Delta x : A$.
- Let $t = adv\, v$. Since Γ' contains a tick, we have that Γ' is of the form $\Gamma_1, \checkmark_{cl(v)}, \Gamma_2$ with Γ_2 tick-free and $\Gamma, \Gamma_1 \vdash_\Delta v : \circledcirc A$. It remains to be shown that $\Gamma^\bigcirc, \Gamma_1 \vdash_\Delta v : \circledcirc A$. The only non-trivial case is when v is a variable x with $x : \circledcirc A \in \Gamma$. Since $\circledcirc A$ is not a stable type, Γ must be of the form $\Gamma_3, x : \circledcirc A, \Gamma_4$ with Γ_4 and Γ_1 tick-free. Since $\circledcirc A$ is a later type, we thus have that $\Gamma^\bigcirc = \Gamma_3^\bigcirc, x : \circledcirc A, \Gamma_4^\bigcirc$, and thus $\Gamma^\bigcirc, \Gamma_1 \vdash_\Delta v : \circledcirc A$.
- The arguments for *select* and $adv_\forall$ are analogous to the argument for *adv* above.
- The cases for *box* and *fix* follow immediately from the fact that $(\Gamma, \Gamma')^\square = (\Gamma^\bigcirc, \Gamma')^\square$.
- The remaining cases follow immediately from the induction hypothesis. The case for *delay* additionally requires the property that $\Gamma, \Gamma' \vdash_\Delta \theta : \mathit{Clock}$ implies $\Gamma^\bigcirc, \Gamma' \vdash_\Delta \theta : \mathit{Clock}$, which follows by a straightforward induction on θ. ∎

Next, we need to establish the closure properties of the logical relations that allow us to prove their fundamental property. The central property of a Kripke logical relation is preservation under the Kripke preorder $\lesssim^\Delta$:

Lemma 5.3. *Let* $w \lesssim^\Delta w'$.

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

Proof (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). ∎

In addition, we have that the term relation subsumes the value relation:

Lemma 5.4. $\mathcal{V}_\Delta[\![A]\!]\,(w) \subseteq \mathcal{T}_\Delta[\![A]\!]\,(w)$.

Proof Let $t \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$, $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$, and $\vdash \iota : \Delta$. By inspection of the definition of $\mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$, we obtain that t is a value and thus $\langle t; \sigma' \rangle \Downarrow^\iota \langle t; \sigma' \rangle$. By Lemma 5.3, we have that $t \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma')$. ∎

Intuitively speaking, stable types are time independent, i.e., values of stable types can be used at any time in the future. This intuition is confirmed by the following property that states that the value relation for stable types is independent of the store σ:

Lemma 5.5. $\mathcal{V}_\Delta[\![A]\!]\,(n, \sigma) = \mathcal{V}_\Delta[\![A]\!]\,(n, \emptyset)$ *for any stable type* A.

Proof By straightforward induction on the size of A. ∎

The reactive semantics performs garbage collection after each output step. To show that this is sound, we need that the logical relation is closed under performing such garbage collection on the store σ. However, we only need this property for elements of the language that can be moved across time steps, namely for values of stable types, which can be moved into the future unchanged, and values of types of the form $\circledcirc A$, which can be advanced in the appropriate next time step. That is, the logical relation need only be closed under garbage

collection for *later types*. To account for this also at the level of typing contexts, we further extend the notion of later types to contexts. We thus call any tick-free context a *later context* if it only contains variables $x : A$ where A is a later type.

Lemma 5.6 (garbage collection)**.**

(i) $\mathcal{V}_\Delta[\![A]\!]\,(n,\sigma) \subseteq \mathcal{V}_\Delta[\![A]\!]\,(n, gc(\sigma))$ *if* A *is a later type.*
(ii) $\mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma) \subseteq \mathcal{C}_\Delta[\![\Gamma]\!]\,(n, gc(\sigma))$ *if* Γ *is a later context.*

Proof If A is a stable type, then (i) follows immediately from Lemma 5.5. Otherwise, if $A = \circledcirc B$, then (i) follows immediately from the definition of the value relation. (ii) follows from a simple induction argument on the length of Γ using (i). ∎

The final lemma in preparation for the fundamental property states that the clock independence property holds for both the value and context relations:

Lemma 5.7.

(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* $[\sigma]_{cl(v)} = [\sigma']_{cl(v)}$.
(ii) If $\gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta]\!]\,(n,\sigma)$*, then* $\gamma \in \mathcal{C}_\Delta[\![\Gamma, \checkmark_\theta]\!]\,\left(n, [\sigma]_{|\theta\gamma|}\right)$.

Proof Both statements 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 clock independence property for $\mathcal{V}_\Delta[\![\circledcirc A]\!]\,(n,\sigma)$ above is a crucial component in the soundness proof for *adv* and *select* in Theorem 5.8 below, where we make use of the fact that *adv* and *select* are only applied to values so that Lemma 5.7 (i) applies.

Finally, the closure properties established above allow us to prove the soundness of the language in the form of the following fundamental property of the logical relation:

Theorem 5.8. *If* $\Gamma \vdash_\Delta$, $\Gamma \vdash_\Delta t : A$*, and* $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma)$*, then* $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n,\sigma)$.

Proof The proof proceeds by structural induction on t. If $t\gamma$ is a value, it suffices to show that $t\gamma \in \mathcal{V}_\Delta[\![A]\!]\,(n,\sigma)$, according to Lemma 5.4. In all other cases, to prove $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n,\sigma)$, we assume some input buffer $\vdash \iota : \Delta$ and store $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$, and show that there exists σ'' and v such that $\langle t\gamma; \sigma'\rangle \Downarrow_\iota \langle v; \sigma''\rangle$ and $v \in \mathcal{V}_\Delta[\![A]\!]\,(n,\sigma'')$. By Lemma 5.3 we may assume that $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma')$.

For the case of $\Gamma \vdash_\Delta \mathit{delay}_\theta\, t : \circledcirc A$, Lemma 5.2 allows us to assume $\Gamma^\bigcirc, \checkmark_\theta \vdash_\Delta t : A$ rather than $\Gamma, \checkmark_\theta \vdash_\Delta t : A$. In turn, this allows us to invoke Lemma 5.6 (ii), which applies to $\Gamma^\bigcirc$ but not necessarily Γ.

The induction proof on t makes use of the aforementioned closure properties of the logical relations. We include the full details of this induction argument 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, namely Theorem 4.1, Proposition 4.4, and Theorem 4.5.

5.2.1 Productivity

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

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

Intuitively speaking, I_n is the set of initial machine states from which the machine can safely run for n steps, and S_n is the set of machine states from which the machine can safely run for n more steps. In the definition of I_n and S_n as well as the proofs below we make use of the notions of well-typedness for input buffers, input values, and output values, which are given at the beginning of section 4.4.1. The following lemma shows that the typing of closed values coincides with the value relation for value types:

Lemma 5.9. *Let A be a value type. Then $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$ iff $\vdash v : A$.*

Proof Straightforward induction on A. ∎

This fact can be used together with the fundamental property to prove the following lemma, which essentially states that the machine stays inside the sets of states I_n and S_n defined above and will only produce well-typed outputs.

Lemma 5.10 (productivity)**.**

(i) If $t : \Delta \Rightarrow \Gamma_{out}$ and $\vdash \iota : \Delta$, 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 $\langle t; \iota \rangle \overset{O}{\Longrightarrow} \langle D; \eta; \iota \rangle$ such that $\langle D; \eta; \iota \rangle \in S_n$ and $\vdash O : \Gamma_{out}$.

(iii) If $\langle D; \eta; \iota \rangle \in S_{n+1}$ and $\vdash \kappa \mapsto v : \Delta$, then there is a sequence of two transitions $\langle D; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle D; \sigma; \iota' \rangle \overset{O}{\Longrightarrow} \langle D'; \eta'; \iota' \rangle$ such that $\langle D'; \eta'; \iota' \rangle \in S_n$ and $\vdash O : \Gamma_{out}$.

Proof

(i) We need to show that $t \in \mathcal{T}_\Delta[\![\mathit{Sigs}\,(\Gamma_{out})]\!]\,(n, \emptyset)$. Since $t : \Delta \Rightarrow \Gamma_{out}$, we know that $\vdash_\Delta t : \mathit{Sigs}\,(\Gamma_{out})$. Hence, by Theorem 5.8, we have that $t \in \mathcal{T}_\Delta[\![\mathit{Sigs}\,(\Gamma_{out})]\!]\,(n, \emptyset)$.

(ii) Let $\langle t; \iota \rangle \in I_n$. That means we have $t \in \mathcal{T}_\Delta[\![\mathit{Sigs}\,(\Gamma_{out})]\!]\,(n, \emptyset)$. Hence, we have that $\langle t; \emptyset \rangle \Downarrow^\iota \langle (v_1 :: w_1, \ldots, v_k :: w_k); \eta \rangle$ with $v_i \in \mathcal{V}_\Delta[\![A_i]\!]\,(n, \eta)$ and $w_i \in \mathcal{V}_\Delta[\![\textcircled{\exists}(\mathit{Sig}\,A_i)]\!]\,(n, \eta)$. Since $\mathit{Sig}\,A_i$ is not a value type, w_i cannot be of the form wait_κ. Hence, by the definition of the value relation, each w_i must be some heap location l_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 $\vdash x_1 \mapsto v_1, \ldots, x_k \mapsto v_k : \Gamma_{out}$.

(iii) By definition, we have $\langle D; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle D; \sigma; \iota' \rangle$, where $\sigma = [\eta]_{\kappa \in} \langle \kappa \mapsto v \rangle\, [\eta]_{\kappa \notin}$, $\iota' = \iota[\kappa \mapsto v]$ if $\kappa \in \mathit{dom}\,(\iota)$, and $\iota' = \iota$ otherwise. To construct the second transition step, we show the following more general property:

$$\begin{aligned} &\text{If } \sigma' \sqsupseteq \sigma, D_1 \subseteq D, \text{ then } \langle D_1; \sigma'; \iota' \rangle \overset{O}{\Longrightarrow} \langle D_2; \eta'; \iota' \rangle, \\ &\text{such that } \langle D_2; \eta'; \iota' \rangle \in S_n, gc(\sigma') \sqsubseteq \eta', \text{ and } \vdash O : \Gamma_{out}. \end{aligned} \tag{3}$$

From (3), we then obtain that $\langle D; \sigma; \iota' \rangle \overset{O}{\Longrightarrow} \langle D'; \eta'; \iota' \rangle$, with $\langle D'; \eta'; \iota' \rangle \in S_n$ and $\vdash O : \Gamma_{out}$. We conclude this proof by proving (3) by induction on the size of D_1. Since $\sigma' \sqsupseteq \sigma$, we know that $\sigma' = \eta_N \langle \kappa \mapsto v \rangle \eta_L$ with $\eta_N \sqsupseteq [\eta]_{\kappa \in}$ and $\eta_L \sqsupseteq [\eta]_{\kappa \notin}$.

- Let $D_1 = \cdot$. Then, by definition, we get the transition $\langle \cdot; \sigma'; \iota' \rangle \stackrel{\cdot}{\Longrightarrow} \langle \cdot; gc(\sigma'); \iota' \rangle$, where the conditions $\langle \cdot; gc(\sigma'); \iota' \rangle \in S_n$, $gc(\sigma') \sqsubseteq gc(\sigma')$, and $\vdash \cdot : \Gamma_{out}$ are trivially true.
- Let $D_1 = x \mapsto l, D_1'$ with $\kappa \notin cl(l)$ and $x : A \in \Gamma_{out}$. By induction hypothesis, we obtain a transition $\left\langle D_1'; \sigma'; \iota' \right\rangle \stackrel{O}{\Longrightarrow} \left\langle D_2'; \eta'; \iota' \right\rangle$, with $\left\langle D_2'; \eta'; \iota' \right\rangle \in S_n$, $gc(\sigma') \sqsubseteq \eta'$, and $\vdash O : \Gamma_{out}$. By definition, we obtain the transition $\left\langle x \mapsto l, D_1'; \sigma'; \iota' \right\rangle \stackrel{O}{\Longrightarrow} \left\langle x \mapsto l, D_2'; \eta'; \iota' \right\rangle$. Since $\langle D; \eta; \iota \rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n+1, \eta)$. Because of $\kappa \notin cl(l)$, we can conclude that $[\eta]_{cl(l)} = \left[[\eta]_{\kappa\notin}\right]_{cl(l)}$, which means that we can apply Lemma 5.7 to obtain that $l \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n+1, [\eta]_{\kappa\notin})$. Since $[\eta]_{\kappa\notin} \sqsubseteq \eta_L = gc(\sigma') \sqsubseteq \eta'$, we have by Lemma 5.3 that $l \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n, \eta')$ and thus $\left\langle x \mapsto l, D_2'; \eta'; \iota' \right\rangle \in S_n$.
- Let $D_1 = x \mapsto l, D_1'$ with $\kappa \in cl(l)$ and $x : A \in \Gamma_{out}$. Since $\langle D; \eta; \iota \rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n+1, \eta)$, which implies $adv\, l \in \mathcal{T}_\Delta[\![Sig\, A]\!]\,(n, \sigma)$ by definition. Hence, by definition, we obtain $\langle adv\, l; \sigma' \rangle \Downarrow^{\iota'} \langle v' :: l'; \sigma'' \rangle$ with $v' \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$ and $l' \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n, \sigma'')$. By induction hypothesis, we obtain a transition $\left\langle D_1'; \sigma''; \iota' \right\rangle \stackrel{O}{\Longrightarrow} \left\langle D_2'; \eta'; \iota' \right\rangle$, with $\left\langle D_2'; \eta'; \iota' \right\rangle \in S_n$, $gc(\sigma'') \sqsubseteq \eta'$, and $\vdash O : \Gamma_{out}$. Since $\sigma' \sqsubseteq \sigma''$ by Lemma 5.1, we also have $gc(\sigma') \sqsubseteq gc(\sigma'') \sqsubseteq \eta'$. By definition, we obtain the transition $\left\langle x \mapsto l, D_1'; \sigma'; \iota' \right\rangle \stackrel{x\mapsto v', O}{\Longrightarrow} \left\langle x \mapsto l', D_2'; \eta'; \iota' \right\rangle$. By Lemma 5.3 and Lemma 5.6, we have that $l' \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(n, \eta')$, and thus we also have $\left\langle x \mapsto l', D_2'; \eta'; \iota' \right\rangle \in S_n$. In addition, since $v' \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$ implies $\vdash v' : A$ by Lemma 5.9, we have that $\vdash x \mapsto v', O : \Gamma_{out}$. ∎

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

Proof [Theorem 4.1 (productivity)] For each $n \in \mathbb{N}$, we can construct the following finite transition sequence s_n using Lemma 5.10:

$$\langle t; \iota_0 \rangle \stackrel{O_0}{\Longrightarrow} \langle D_0; \eta_0; \iota_0 \rangle \stackrel{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle D_0; \sigma_0; \iota_1 \rangle \stackrel{O_1}{\Longrightarrow} \langle D_1; \eta_1; \iota_1 \rangle \stackrel{\kappa_1 \mapsto v_1}{\Longrightarrow} \ldots \stackrel{O_n}{\Longrightarrow} \langle D_n; \eta_n; \iota_n \rangle$$

with $\vdash O_i : \Gamma_{out}$ 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*

Also the signal independence property for buffered signals is a straightforward consequence of Lemma 5.10:

Proof [Proposition 4.4 (buffered signal independence)] By Lemma 5.10, the state $\langle D_i; \eta_i; \iota_i \rangle$ of any input transition $\langle D_i; \eta_i; \iota_i \rangle \stackrel{\kappa_i \mapsto v_i}{\Longrightarrow} \langle D_i; \sigma_i; \iota_{i+1} \rangle$ in a sequence starting from $\langle t; \iota_0 \rangle$ will be in S_1. In particular, this means that $l \in \mathcal{V}_\Delta[\![\circledcirc(Sig\, A)]\!]\,(1, \eta_i)$ for any $x \mapsto l \in D_i$ with $x : A \in \Gamma_{out}$, which in turn means that $l \in Loc_\Delta$. Hence, $\kappa_i \notin cl(l)$, and so O_{i+1} is empty.

By the definition of input transitions, $\sigma_i = [\eta_i]_{\kappa_i \in} \langle \kappa_i \mapsto v_i \rangle\, [\eta_i]_{\kappa_i \notin}$. Since κ_i is buffered-only, it cannot be a member of any clock. Consequently, $[\eta_i]_{\kappa_i \in} = \emptyset$ and $[\eta_i]_{\kappa_i \notin} = \eta_i$, which means that $\sigma_i = \emptyset\, \langle \kappa_i \mapsto v_i \rangle\, \eta_i$.

We show $D_{i+1} = D_i$ and $\eta_{i+1} = \eta_i$ by induction on the derivation of the output transition $\langle D_i; \sigma_i; \iota_{i+1} \rangle \stackrel{O_{i+1}}{\Longrightarrow} \langle D_{i+1}; \eta_{i+1}; \iota_{i+1} \rangle$. Since O_{i+1} is empty, the derivation of the output transition cannot use OUTPUT-COMPUTE. If the output transition is due to OUTPUT-END, then

$D_{i+1} = D_i$ follows immediately, and $\eta_{i+1} = \eta_i$ follows from the above established fact that $\sigma_i = \emptyset\,\langle \kappa_i \mapsto v_i \rangle\,\eta_i$. If the output transition is due to OUTPUT-SKIP, then both $D_{i+1} = D_i$ and $\eta_{i+1} = \eta_i$ follow immediately by the induction hypothesis. ∎

For the proof of Theorem 4.5, we first define the following sets of machine states in the context of a partial map $\overline{\Delta}$ that maps variables x to input contexts Δ_x:

$$T_n^{\overline{\Delta}} = \left\{ \langle D; \eta; \iota \rangle \;\middle|\; \begin{array}{l} \vdash \iota : \Delta \;\wedge\; \forall x \mapsto w \in D.\Delta_x \subseteq \Delta \\ \wedge\;\; \exists x : A \in \Gamma_{out}.w \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\eta) \end{array} \right\}$$

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

Lemma 5.11. *If* $\langle D; \eta; \iota \rangle \in T_{n+1}^{\overline{\Delta}}$ *and* $\langle D; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle D; \sigma; \iota' \rangle \overset{O}{\Longrightarrow} \langle D'; \eta'; \iota' \rangle$, *then* $\langle D'; \eta'; \iota' \rangle \in T_n^{\overline{\Delta}}$.

Proof Note that $\sigma = [\eta]_{\kappa\in}\,\langle \kappa \mapsto v \rangle\,[\eta]_{\kappa\notin}$ and that $[\eta]_{\kappa\notin} \sqsubseteq \eta'$. Suppose $x \mapsto w \in D$, and note that w must be some location l, since $Sig\ A$ is not a value type. Then there is an l' such that $x \mapsto l' \in D'$, and we must show that $l' \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\eta')$ using the hypothesis $l \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n+1,\eta)$. Suppose first that $\kappa \notin cl(l)$. Then $l' = l$ and since $\kappa \notin cl(l)$, we know that $[\eta]_{cl(l)} = \left[[\eta]_{\kappa\notin}\right]_{cl(l)}$, which, by Lemma 5.7, means that $l' \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,\left(n+1, [\eta]_{\kappa\notin}\right)$. We can then apply Lemma 5.3 to conclude that $l' \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\eta')$.

Suppose now $\kappa \in cl(l)$. In that case, l' must have arisen from an evaluation $\langle adv\ l; \sigma' \rangle \Downarrow^{\iota} \langle w' :: l'; \sigma'' \rangle$ for some σ', σ'' such that $\sigma \sqsubseteq \sigma'$, and $gc(\sigma'') \sqsubseteq \eta'$ as well as $l' \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\sigma'')$. The hypothesis tells us that

$$\eta(l) \in \mathcal{T}_{\Delta_x}[\![Sig\ A]\!]\,\left(n, [\eta]_{\kappa\in}\,\langle \kappa \mapsto v \rangle\,[\eta]_{\kappa\notin}\right)$$

Since $[\eta]_{\kappa\in}\,\langle \kappa \mapsto v \rangle\,[\eta]_{\kappa\notin} \sqsubseteq \sigma'$, this means that $\langle \eta(l); \sigma' \rangle \Downarrow^{\iota} \langle w'' :: l''; \sigma''' \rangle$. Since the now heap of σ' maps l to $[\eta]_{\kappa\in}\,(l) = \eta(l)$, evaluating $adv\ l$ amounts to evaluating $\eta(l)$, and thus by Lemma 4.2, $\sigma''' = \sigma''$ and $l'' = l'$. From this we conclude that

$$l' \in \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\sigma'') \subseteq \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n, gc(\sigma'')) \subseteq \mathcal{V}_{\Delta_x}[\![\textcircled{\exists}(Sig\ A)]\!]\,(n,\eta')$$

where the first inclusion follows from Lemma 5.6 and the second inclusion follows from Lemma 5.3. ∎

With the help of Lemma 5.11, we can now prove the push signal independence property:

Proof [Theorem 4.5 (push signal independence)] The initialisation transition

$$\langle (t_1, \ldots, t_m)\,; \iota_0 \rangle \overset{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m}{\Longrightarrow} \langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m; \eta_0; \iota_0 \rangle = \langle D_0; \eta_0; \iota_0 \rangle$$

is caused by an evaluation of the form $\langle (t_1, \ldots, t_m)\,; \emptyset \rangle \Downarrow^{\iota} \langle (v_1 :: l_1, \ldots, v_m :: l_m)\,; \eta_0 \rangle$, which in turn means that $\langle t_j; \eta' \rangle \Downarrow^{\iota} \langle v_j :: l_j; \eta'' \rangle$ for some η', η'' with $\eta'' \sqsubseteq \eta_0$ by Lemma 5.1. By assumption $\vdash_{\Delta'} t_j : Sig\ A_j$ and thus $t_j \in \mathcal{T}_{\Delta'}[\![Sig\ A_j]\!]\,(n,\emptyset)$ for all n by Theorem 5.8, which in turn implies $l_j \in \mathcal{V}_{\Delta'}[\![\textcircled{\exists}(Sig\ A_j)]\!]\,(n,\eta_0)$ for all n. Likewise, $l_k \in \mathcal{V}_{\Delta}[\![\textcircled{\exists}(Sig\ A_k)]\!]\,(n,\eta_0)$ for all n, and $k = 1, \ldots, m$. So $\langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m; \eta_0; \iota_0 \rangle \in T_n^{\overline{\Delta}}$ for all n, where $\Delta_{x_j} = \Delta'$ and $\Delta_{x_k} = \Delta$, for $k \neq j$. By i applications of Lemma 5.11, we thus obtain that $\langle D_i; \eta_i; \iota_i \rangle \in T_1^{\overline{\Delta}}$. In particular, $D_i(x_j) = l'$ for some $l' \in \mathcal{V}_{\Delta'}[\![\textcircled{\exists}(Sig\ A_j)]\!]\,(1,\eta_i)$. Hence, $cl(l') \subseteq dom\,(\Delta')$ and thus $x_j \mapsto v \in O_{i+1}$ implies $\kappa_i \in cl(l')$, which in turn implies $\kappa_i \in dom\,(\Delta')$. ∎

$$\frac{\Gamma \Vdash_\Delta t : \textcircled{\exists} A}{\Gamma \Vdash_\Delta cl(t) : Clock} \qquad \frac{\Gamma \Vdash_\Delta t : \textcircled{\exists} A \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_{cl(t)}, \Gamma' \Vdash_\Delta adv\, t : A}$$

$$\frac{\Gamma \Vdash_\Delta t_1 : \textcircled{\exists} A_1 \qquad \Gamma \Vdash_\Delta t_2 : \textcircled{\exists} A_2 \qquad \theta = cl(t_1) \sqcup cl(t_2) \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta select\, t_1\, t_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)}$$

Fig. 9. Revised typing rules for *Full Async RaTT*.

6 Full Async RaTT

In this section, we give a slightly more general type system that eliminates the value restriction for clocks and for arguments of *adv* and *select*. This generalised type system, which we refer to as *Full Async RaTT*, allows both *adv* and *select* to be applied to arbitrary terms of the appropriate type rather than just values.

The purpose of *Full Async RaTT* is to demonstrate that the value restriction for *adv* and *select* does not limit the expressiveness of the *Async RaTT* calculus. At the same time, this restriction *is* crucial to obtain the operational properties proved in section 5: As we shall see in section 6.3, it is not sound to apply the operational semantics of *Async RaTT* directly to *Full Async RaTT*. However, we will show that we can transform any well-typed, closed *Full Async RaTT* term into a uniquely defined well-typed *Async RaTT* term of the same type, which can then safely run on the machine presented in section 4. This program transformation is also used in the implementation of *Full Async RaTT* as an embedded language in Haskell (Bahr et al. 2024a).

Full Async RaTT shares almost all of its typing rules with *Async RaTT*; the only typing rules that change are those marked with ⋆ in Figure 3. The new typing rules that replace these in *Full Async RaTT* are shown in Figure 9. To distinguish the type system from that of Figure 3, we use the symbol $\Vdash$ instead of $\vdash$.

6.1 Program transformation

In this section, we present a program transformation that turns any *Full Async RaTT* program into an *Async RaTT* program of the same type. This program transformation accounts for the fact that *Full Async RaTT* allows *adv* and *select* to be applied to non-value terms. The program transformation is formalised as a rewrite relation $\longrightarrow$ on terms defined by the rewrite rules in Figure 10. In the following, we will explain the idea behind the rewrite relation as well as the terminology we use to define it, in particular the notion of *tick-closed* terms and the *tick-substitution* operation $t(\!|s|\!)$.

All rewrite rules replace occurrences of subterms of the form $delay_\theta\, s$, where θ contains a non-value term, and where s is already in normal form, i.e., no rewrite rule can be applied to s. The side condition on s is crucial to ensure that rewriting is type preserving. Rule (T1) considers the case where s *may* contain an occurrence of *adv* that consumes the tick $\checkmark_\theta$ introduced by $delay_\theta$ into the typing context. This can only happen if θ is of the form $cl(t)$ for some term t. Rules (T2) and (T3) perform an analogous transformation, but for *select* instead of *adv*. Finally, rule (T4) considers the case where s does not contain any occurrence of *adv* or *select* that consumes the tick $\checkmark_\theta$. Note that the rules are not mutually exclusive,

$$delay_{cl(t)}\ s \longrightarrow let\, x = t\, in\, delay_{cl(x)}\,(s(\!|adv\, x|\!)) \quad \text{if } t \text{ is not a value} \tag{T1}$$

$$delay_{cl(t_1)\sqcup cl(t_2)}\ s \longrightarrow let\, x = t_1\, in\, delay_{cl(x)\sqcup cl(t_2)}\,(s(\!|select\, x\, t_2|\!)) \quad \text{if } t_1 \text{ is not a value} \tag{T2}$$

$$delay_{cl(v)\sqcup cl(t)}\ s \longrightarrow let\, y = t\, in\, delay_{cl(v)\sqcup cl(y)}\,(s(\!|select\, v\, y|\!)) \quad \text{if } t \text{ is not a value} \tag{T3}$$

$$delay_{T[cl(t)]}\ s \longrightarrow let\, x = t\, in\, delay_{T[cl(x)]}\ s \quad \text{if } s \text{ is tick-closed and } t \text{ not a value} \tag{T4}$$

For all rules, we have the side condition that s is in normal form.

Fig. 10. Rewrite rules for the transformation of *Full Async RaTT* terms to *Async RaTT* terms.

e.g., (T1) and (T4) may apply if $\theta = cl(t)$ and s does not contain any corresponding *adv*. Similarly, both (T2) and (T3) may apply simultaneously with (T4).

To see the rewrite rules in action, consider the closed *Full Async RaTT* term

$$\lambda x.delay_{cl(x)}\ (delay_{cl(adv\, x)}\ (suc\ (adv\ (adv\, x)))) \tag{4}$$

of type $\Circled{\exists}(\Circled{\exists}Nat) \to \Circled{\exists}(\Circled{\exists}Nat)$. It defines a function that increments a doubly delayed number. This term is not well-typed in *Async RaTT* since *adv* is applied to the non-value term $adv\, x$. However, we can transform this term into a well-typed *Async RaTT* term that first evaluates $adv\, x$ to a value y and then applies *adv* to y instead of $adv\, x$:

$$\lambda x.delay_{cl(x)}\ (let\, y = adv\, x\, in\, delay_{cl(y)}\ (suc\ (adv\, y))) \tag{5}$$

Rule (T1) captures exactly this transformation. To this end, the rewrite rule uses the tick-substitution operation $t(\!|s|\!)$ that replaces all occurrences of *adv* and *select* in t that are not guarded by *delay*, *box*, or *fix* with s:

$$\begin{aligned} t(\!|s|\!) &= t && \text{if } t \text{ is of the form } delay\, t', box\, t', \text{ or } fix\, y.t' \\ t(\!|s|\!) &= s && \text{if } t \text{ is of the form } adv\, t' \text{ or } select\, t_1\, t_2 \\ (t_1\, t_2)(\!|s|\!) &= (t_1(\!|s|\!))\ (t_2(\!|s|\!)) && \text{and similarly for all other terms} \end{aligned}$$

Rule (T1) uses tick-substitution to replace occurrences of $adv\, t$ in s with $adv\, x$: Well-typing of the term $delay_{cl(t)}\ s$ ensures that s does not contain *select* and that all occurrences of *adv* are of the form $adv\, t$. Returning to the example term (4), we can see that we obtain the term (5) by performing the following substitution:

$$(suc\ (adv\ (adv\, x)))(\!|adv\, y|\!) = suc\ (adv\, y)$$

The same idea is used to account for *select* instead of *adv*. However, we have two rules, (T2) and (T3) that handle each of the two arguments of *select* separately.

Finally, we turn to rule (T4), which considers the case where $delay_\theta$ is applied to a term that has no occurrences of *adv* or *select* that consume the tick on θ. For example, consider the following *Full Async RaTT* term of type $(Nat \to \Circled{\exists}1) \to \Circled{\exists}\,(\Circled{\exists}Nat \to \Circled{\exists}Nat)$:

$$\lambda x.delay_{cl(x\ 0)\,\sqcup\, cl(x\ 1)\,\sqcup\, cl(x\ 2)}\ (\lambda y.delay_{cl(y)}\ (suc\ (adv\, y))) \tag{6}$$

The clock expression $cl(x\ 0) \sqcup cl(x\ 1) \sqcup cl(x\ 2)$ is not well-formed in *Async RaTT* since it contains terms that are not values. Because the argument to $delay_{cl(x\ 0)\sqcup cl(x\ 1)\sqcup cl(x\ 2)}$ has no occurrence of *adv* or *select* that consumes the tick of clock $cl(x\ 0) \sqcup cl(x\ 1) \sqcup cl(x\ 2)$, the clock expression is not of the right form for any of the first three rules to be applicable.

Rule (T4) covers exactly this case. Repeatedly applying this rule transforms (6) into a term that first evaluates $x\ 0$, $x\ 1$, and $x\ 2$ to values before constructing the clock:

$$\lambda x.\mathit{let}\ z_0 = x\ 0\ \mathit{in}\ \mathit{let}\ z_1 = x\ 1\ \mathit{in}\ \mathit{let}\ z_2 = x\ 2\ \mathit{in}$$
$$\mathit{delay}_{cl(z_0)\sqcup cl(z_1)\sqcup cl(z_2)}\ (\lambda y.\mathit{delay}_{cl(y)}\ (\mathit{suc}\ (\mathit{adv}\ y)))$$

To understand rewrite rule (T4), we need to introduce some terminology: We use the notation T for a clock expression with a single hole $[]$ such that for all clock expressions of the form $cl(t)$ occurring to the left of the hole $[]$, we have that t is a value. More precisely, T is generated by the following grammar:

$$T ::= [] \mid T \sqcup \theta \mid \overline{\theta} \sqcup T$$
$$\overline{\theta} ::= cl(v) \mid \overline{\theta} \sqcup \overline{\theta}$$

We say that a clock expression of the form $\overline{\theta}$ is *simple*, and we write $T[\theta]$ for the clock expression obtained from T by replacing the unique hole in T by θ. For example, if $T = cl(x) \sqcup []$, then $T[cl(y)] = cl(x) \sqcup cl(y)$. Similarly, we use the notation K for a term with a single occurrence of a hole $[]$ that must not be in the scope of *delay*, *adv*, *select*, *box*, or *fix*. More precisely, K is generated by the following grammar:

$$\begin{aligned} K ::= {} & [] \mid K\,t \mid t\,K \mid \lambda x.K \mid \mathit{let}\,x = K\,\mathit{in}\,t \mid \mathit{let}\,x = t\,\mathit{in}\,K \mid (K,t) \mid (t,K) \mid \mathit{in}_1\,K \mid \mathit{in}_2\,K \\ & \mid \pi_1\,K \mid \pi_2\,K \mid \mathit{case}\,K\,\mathit{of}\,\mathit{in}_1\,x.s; \mathit{in}_2\,x.t \mid \mathit{case}\,s\,\mathit{of}\,\mathit{in}_1\,x.K; \mathit{in}_2\,x.t \\ & \mid \mathit{case}\,s\,\mathit{of}\,\mathit{in}_1\,x.t; \mathit{in}_2\,x.K \mid \mathit{suc}\,K \mid \mathit{into}\,K \mid \mathit{out}\,K \mid \mathit{unbox}\,K \mid \mathit{rec}_{Nat}(K, x\,y.t, u) \\ & \mid \mathit{rec}_{Nat}(s, x\,y.K, u) \mid \mathit{rec}_{Nat}(s, x\,y.t, K) \end{aligned}$$

We write $K[t]$ for the term obtained from K by substituting the unique hole $[]$ in K with the term t. Finally, we say that a term t is *tick-closed* iff t is neither of the form $K[\mathit{adv}\,s]$ nor $K[\mathit{select}\,s\,s']$. In other words, a term t is tick-closed iff any occurrence of *adv* and *select* in t is in the scope of *delay*, *box*, or *fix*. Tick substitution is closely related to contexts of the form K. Namely, $t(\!|s|\!)$ is the result of repeatedly applying to t the rewrite relation defined by $K[\mathit{adv}\,s'] \longrightarrow K[s]$ and $K[\mathit{select}\,s_1\,s_2] \longrightarrow K[s]$.

Returning to the example, we can see that (T4) can be applied to (6) since the term $\lambda y.\mathit{delay}_{cl(y)}\ (\mathit{suc}\ (\mathit{adv}\ y))$ is tick-closed: The occurrence of $\mathit{adv}\ y$ is guarded by $\mathit{delay}_{cl(y)}$ and thus $\lambda y.\mathit{delay}_{cl(y)}\ (\mathit{suc}\ (\mathit{adv}\ y))$ is not of the form $K[\mathit{adv}\ y]$.

6.2 Soundness of the program transformation

We show that the rewrite relation $\longrightarrow$ is strongly normalising, confluent, and type-preserving in the *Full Async RaTT* calculus. Moreover, any closed *Full Async RaTT* term that cannot be further rewritten is also well-typed in *Async RaTT*. That is, by exhaustively applying the rewrite rules to a closed *Full Async RaTT* term, we can transform it into a unique *Async RaTT* term of the same type:

Theorem 6.1. *For each* $\Vdash_\Delta t : A$*, we can effectively construct a unique term* t' *with* $t \longrightarrow^* t'$ *and* $\vdash_\Delta t' : A$.

The proof of the above theorem can be broken down into the following four properties, where we use the notation $t \not\longrightarrow$ to denote that there is no t' with $t \longrightarrow t'$:

(1) Confluence: If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then $t_1 \longrightarrow^* s$ and $t_2 \longrightarrow^* s$ for some s.

(2) Exhaustiveness: If $t \not\longrightarrow$, then $\Vdash_\Delta t : A$ implies $\vdash_\Delta t : A$.
(3) Subject reduction: If $\Gamma \Vdash_\Delta s : A$ and $s \longrightarrow t$, then $\Gamma \Vdash_\Delta t : A$.
(4) Strong normalisation: There is no infinite $\longrightarrow$-reduction sequence.

Theorem 6.1 follows immediately from these four properties. We prove each of them in turn below.

6.2.1 *Confluence*

As mentioned earlier, sometimes more than one rewrite rule from Figure 10 is applicable. However, in all such cases of overlapping rules, the transformations result in the same term. We can show that the outcome produced by the rewrite system must be unique:

Proposition 6.2. *The rewrite rules in Figure 10 form a confluent rewrite system. That is, if $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then there is some s with $t_1 \longrightarrow^* s$ and $t_2 \longrightarrow^* s$.*

Proof To show that a higher-order rewrite system is confluent it suffices to show that the rules are weakly orthogonal (Terese 2003; van Raamsdonk 1996). A rewrite system is weakly orthogonal if it is left-linear, i.e., no metavariable occurs more than once in the left-hand side of a rule, and all critical pairs are trivial. A critical pair (t_1, t_2) is the result of rewriting a redex with two overlapping rules (or two different but overlapping instantiations of the same rule), i.e., $t \longrightarrow t_1$ and $t \longrightarrow t_2$. A critical pair (t_1, t_2) is trivial if $t_1 = t_2$.

The rewrite system in Figure 10 is not quite a higher-order rewrite system in the sense of van Raamsdonk (1996), since our rewrite rules have side conditions. However, the underlying proof technique of complete developments can be readily extended to our setting, e.g., following van Oostrom (1997), because our rewrite system disallows nested redexes due to the restriction that s is in normal form in all rules.

All four rules (T1)–(T4) are left-linear, so it remains to be shown that all critical pairs are trivial. None of the rules (T1)–(T3) overlap with each other, nor does (T4) overlap with itself. However, (T4) overlaps with each of the rules (T1)–(T3). For example, (T1) and (T4) overlap on the redex $delay_{cl(t)}\ s$ where s is tick-closed and t is not a value. However, the outcome of applying either of these rules is the same, because $s(\!|adv\, x|\!) = s$ whenever s is tick-closed. In other words, the overlap results in a trivial critical pair. The same argument applies to overlaps of (T4) with (T2) and (T3). ∎

6.2.2 *Exhaustiveness*

To prove the exhaustiveness property, we need three lemmas that allow us to show that any clock expression occurring in a term $t \not\longrightarrow$ is indeed also a well-formed clock expression in *Async RaTT*:

Lemma 6.3. *Let $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta K[t] : A$ and Γ' tick-free.*

(i) If $t = adv\, s$, then $\theta = cl(s)$.
(ii) If $t = select\, t_1\, t_2$, then $\theta = cl(t_1) \sqcup cl(t_2)$.

Proof By straightforward induction on K. ∎

Recall that a clock expression θ is called *simple* if it is a $\sqcup$-combination of clock expressions of the form $cl(v)$. Any well-formed simple clock expression in *Full Async RaTT* is also a well-formed clock expression in *Async RaTT*:

Lemma 6.4. *If $\Gamma \Vdash_\Delta \theta : Clock$ and θ is simple, then $\Gamma \vdash_\Delta \theta : Clock$.*

Proof Straightforward induction on $\Gamma \Vdash_\Delta \theta : \mathit{Clock}$. ∎

Moreover, exhaustively applying $\longrightarrow$ ensures that all clock expressions are indeed simple:

Lemma 6.5. *If* $\Gamma, \checkmark_\theta \Vdash_\Delta t : A$ *and* $\mathit{delay}_\theta\, t \not\longrightarrow$, *then* θ *is simple.*

Proof We consider two cases:

- t is tick-closed: Then $\mathit{delay}_\theta\, t \not\longrightarrow$ implies that θ is simple.
- t is not tick-closed: Hence, t is of the form $K[\mathit{adv}\, s]$ or $K[\mathit{select}\, t_1\, t_2]$, which by Lemma 6.3 means that $\theta = \mathit{cl}(s)$ or $\theta = \mathit{cl}(t_1) \sqcup \mathit{cl}(t_2)$, respectively. Hence, $\mathit{delay}_\theta\, t \not\longrightarrow$ implies that θ is simple. ∎

The exhaustiveness property is proved by induction on the typing judgement. To this end, we generalise the property from closed *Full Async RaTT* terms to open *Full Async RaTT* terms in contexts that are also valid in *Async RaTT*:

Proposition 6.6 (exhaustiveness)**.** *If* $\Gamma \vdash_\Delta$, $\Gamma \Vdash_\Delta t : A$, *and* $t \not\longrightarrow$, *then* $\Gamma \vdash_\Delta t : A$.

Proof We proceed by induction on $\Gamma \Vdash_\Delta t : A$:

- $\Gamma \Vdash_\Delta \mathit{delay}_\theta\, t : \Circled{\exists} A$: Hence, $\Gamma, \checkmark_\theta \Vdash_\Delta t : A$ and $\Gamma \Vdash_\Delta \theta : \mathit{Clock}$. By Lemma 6.4 and Lemma 6.5, we have that $\Gamma \vdash_\Delta \theta : \mathit{Clock}$ and thus $\Gamma, \checkmark_\theta \vdash_\Delta$. Consequently, we have $\Gamma, \checkmark_\theta \vdash_\Delta t : A$ by induction and thus $\Gamma \vdash_\Delta \mathit{delay}_\theta\, t : \Circled{\exists} A$.
- $\Gamma, \checkmark_{\mathit{cl}(t)}, \Gamma' \Vdash_\Delta \mathit{adv}\, t : A$: Hence, $\Gamma \Vdash_\Delta t : \Circled{\exists} A$. From the assumption $\Gamma, \checkmark_{\mathit{cl}(t)}, \Gamma' \vdash_\Delta$ we obtain that t is a value and that $\Gamma \vdash_\Delta$. Hence, we may apply the induction hypothesis to obtain that $\Gamma \vdash_\Delta t : \Circled{\exists} A$. Since t is a value, we thus have $\Gamma, \checkmark_{\mathit{cl}(t)}, \Gamma' \vdash_\Delta \mathit{adv}\, t : A$.
- $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta \mathit{select}\, t_1\, t_2 : ((A_1 \times \Circled{\exists} A_2) + (\Circled{\exists} A_1 \times A_2)) + (A_1 \times A_2)$: This follows from an argument analogous to the one above.
- All remaining cases follow immediately from the induction hypothesis, because all other typing rules are the same in *Async RaTT* and *Full Async RaTT*. ∎

6.2.3 Subject reduction

The proof of subject reduction makes use of the side condition that the rewrite rules in Figure 10 can only be applied to a term $\mathit{delay}_\theta\, s$ if $s \not\longrightarrow$. As we have shown in Lemma 6.5, this side condition ensures that all clock expressions in s must be simple, which allows us to apply the following lemma:

Lemma 6.7. *If* $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta t : A$, $\Gamma \Vdash_\Delta \theta' : \mathit{Clock}$, *and* Γ' *contains a tick on a simple clock expression, then* $\Gamma, \checkmark_{\theta'}, \Gamma' \Vdash_\Delta t : A$.

Proof We proceed by induction on $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta t : A$:

- If $t = \mathit{adv}\, s$, then there are Γ_1 and Γ_2 such that Γ_2 is tick-free, $\Gamma' = \Gamma_1, \checkmark_{\mathit{cl}(s)}, \Gamma_2$ and $\Gamma, \checkmark_\theta, \Gamma_1 \Vdash_\Delta s : \Circled{\exists} A$. To conclude $\Gamma, \checkmark_{\theta'}, \Gamma' \Vdash_\Delta t : A$, it remains to be shown that $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \Circled{\exists} A$:
  - If s is a value, then it is either a variable or of the form wait_κ. In either case, we have that $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \Circled{\exists} A$.
  - If s is not a value, then Γ_1 must still contain a tick on a simple clock expression, because $\mathit{cl}(s)$ is not simple. Hence, we can apply the induction hypothesis to obtain $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \Circled{\exists} A$.
- The case $t = \mathit{select}\, t_1\, t_2$ is similar to the previous case.

- If $t = \mathit{box}\, s$, or $t = \mathit{fix}\, x.s$, then $\Gamma, \checkmark_{\theta'}, \Gamma' \Vdash_\Delta t : A$ follows immediately.
- The remaining cases follow immediately from the induction hypothesis. ∎

The essence of the subject reduction property for each of the four rewrite rules is captured by the following lemma:

Lemma 6.8. *Let $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta t : A$, $t \not\longrightarrow$, Γ' tick-free, $\Gamma \Vdash_\Delta \theta : \mathit{Clock}$, and x a fresh variable for t.*

(i) If $\theta = cl(s)$, then $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|\mathit{adv}\, x|\!) : A$ and $\Gamma \Vdash_\Delta s : \textcircled{\exists} B$ for some B.

(ii) If $\theta = cl(t_1) \sqcup cl(t_2)$, then $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x) \sqcup cl(t_2)}, \Gamma' \Vdash_\Delta t(\!|\mathit{select}\, x\, t_2|\!) : A$ and $\Gamma \Vdash_\Delta t_1 : \textcircled{\exists} B$ for some B.

(iii) If $\theta = cl(t_1) \sqcup cl(t_2)$, then $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(t_1) \sqcup cl(x)}, \Gamma' \Vdash_\Delta t(\!|\mathit{select}\, t_1\, x|\!) : A$ and $\Gamma \Vdash_\Delta t_2 : \textcircled{\exists} B$ for some B.

(iv) If $\theta = T[cl(s)]$ and t tick-closed, then $\Gamma, x : \textcircled{\exists} B, \checkmark_{T[cl(x)]}, \Gamma' \Vdash_\Delta t : A$ and $\Gamma \Vdash_\Delta s : \textcircled{\exists} B$ for some B.

Proof We prove (i). The other three statements can be proved in an analogous fashion.
$\Gamma \Vdash_\Delta cl(s) : \mathit{Clock}$ implies that $\Gamma \Vdash_\Delta s : \textcircled{\exists} B$ for some B. We proceed by induction on t:

- If t is of the form $\mathit{box}\, t'$, $\mathit{delay}_{\theta'}\, t'$, or $\mathit{fix}\, y.t'$, then $t(\!|\mathit{adv}\, x|\!) = t$. By weakening, we have $\Gamma, x : \textcircled{\exists} B, \checkmark_\theta, \Gamma' \Vdash_\Delta t(\!|\mathit{adv}\, x|\!) : A$. If t is of the form $\mathit{box}\, t'$ or $\mathit{fix}\, y.t'$, then $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|\mathit{adv}\, x|\!) : A$ follows immediately from the typing rules for *box* and *fix*. Otherwise, if $t = \mathit{delay}_{\theta'}\, t'$, then $\Gamma, x : \textcircled{\exists} B, \checkmark_\theta, \Gamma', \checkmark_{\theta'} \Vdash_\Delta t' : A'$ for some A' with $\textcircled{\exists} A' = A$. Then we can apply Lemma 6.5 to obtain that $\Gamma', \checkmark_{\theta'}$ contains a tick on a simple clock expression. This allows us to apply Lemma 6.7 to obtain that $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)}, \Gamma', \checkmark_{\theta'} \Vdash_\Delta t' : A'$. Hence, $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|\mathit{adv}\, x|\!) : A$ follows.
- The case $t = \mathit{select}\, t_1\, t_2$ is impossible.
- If $t = \mathit{adv}\, s'$, then $t(\!|\mathit{adv}\, x|\!) = \mathit{adv}\, x$ and $A = B$. Hence, $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|\mathit{adv}\, x|\!) : A$.
- The remaining cases all follow by induction hypothesis. ∎

The subject reduction property then follows from the above lemma in a straightforward manner:

Proposition 6.9 (subject reduction)**.** *If $\Gamma \Vdash_\Delta s : A$ and $s \longrightarrow t$, then $\Gamma \Vdash_\Delta t : A$.*

Proof We proceed by induction on $s \longrightarrow t$.

- Let $s \longrightarrow t$ be due to congruence closure. Then $\Gamma \Vdash_\Delta t : A$ follows by the induction hypothesis. For example, if $s = s_1\, s_2$, $t = t_1\, s_2$ and $s_1 \longrightarrow t_1$, then we know that $\Gamma \Vdash_\Delta s_1 : B \to A$ and $\Gamma \Vdash_\Delta s_2 : B$ for some type B. By induction hypothesis, we then have that $\Gamma \Vdash_\Delta t_1 : B \to A$ and thus $\Gamma \Vdash_\Delta t : A$.
- If $s \longrightarrow t$ is due to any of the four rewrite rules, then we can show $\Gamma \Vdash_\Delta t : A$ using Lemma 6.8: For example, in case of rule (T1), we have that $s = \mathit{delay}_{cl(t')}\, s''$, $t = \mathit{let}\, x = t'\, \mathit{in}\, \mathit{delay}_{cl(x)}(s''(\!|\mathit{adv}\, x|\!))$, and $s'' \not\longrightarrow$. Consequently, A is of the form $\textcircled{\exists} A'$ for some A' with $\Gamma, \checkmark_{cl(t')} \Vdash_\Delta s'' : A'$ and $\Gamma \Vdash_\Delta cl(t') : \mathit{Clock}$. By Lemma 6.8, we have that $\Gamma, x : \textcircled{\exists} B, \checkmark_{cl(x)} \Vdash_\Delta s''(\!|\mathit{adv}\, x|\!) : A'$ and $\Gamma \Vdash_\Delta t' : \textcircled{\exists} B$ for some B. Hence, the typing rules for *delay* and let bindings apply to yield $\Gamma \Vdash_\Delta t : A$. ∎

6.2.4 *Strong normalisation*

Finally, strong normalisation of the rewrite relation follows from the observation that each rewrite step removes at least one occurrence of a non-simple clock expression $cl(t)$:

Proposition 6.10 (strong normalisation)**.** *The rewrite relation $\longrightarrow$ is strongly normalising on well-typed Full Async RaTT terms.*

Proof To show that $\longrightarrow$ is strongly normalising, we define for each term u a natural number $d(u)$ such that, whenever $\Gamma \Vdash_\Delta u : A$ and $u \longrightarrow u'$, then $d(u) > d(u')$. By Proposition 6.9, also $\Gamma \Vdash_\Delta u' : A$ and thus strong normalisation follows. Let $d(t)$ be defined as the number of occurrences of clock expressions of the form $cl(s)$ in t where s is not a value, i.e.:

$$d(\theta_1 \sqcup \theta_2) = d(\theta_1) + d(\theta_2) \qquad d(cl(v)) = 0 \qquad d(cl(t)) = 1 + d(t) \text{ if } t \text{ is not a value.}$$
$$d(delay_\theta\, t) = d(\theta) + d(t) \qquad d(s\,t) = d(s) + d(t) \qquad d(\lambda x.t) = d(t) \qquad \text{etc.}$$

We proceed by induction on $u \longrightarrow u'$:

- If $u \longrightarrow u'$ is by congruence closure, then $d(u) > d(u')$ follows by induction.
- If $u = delay_{cl(t)}\, s$ and $u' = let\, x = t\, in\, delay_{cl(x)}\, (s(\!|adv\, x|\!))$ where t is not a value, then $d(u) = d(t) + d(s) + 1$ and $d(u') = d(t) + d(s(\!|adv\, x|\!))$. Since $d(adv\, x) = 0$, we have that $d(s(\!|adv\, x|\!)) \leq d(s)$, and thus we have the desired decrease $d(u) > d(u')$.
- If $u = delay_{cl(t_1) \sqcup cl(t_2)}\, s$ and $u' = let\, x = t_1\, in\, delay_{cl(x) \sqcup cl(t_2)}\, (s(\!|select\, x\, t_2|\!))$, where t_1 is not a value, then $d(u) = d(t_1) + d(cl(t_2)) + d(s) + 1$ and $d(u') = d(t_1) + d(cl(t_2)) + d(s(\!|select\, x\, t_2|\!))$. Since $\Gamma, \checkmark_{cl(t_1) \sqcup cl(t_2)}, \Gamma' \Vdash_\Delta s : A$ for some A and tick-free Γ', we can use Lemma 6.3 to observe that $s(\!|select\, x\, t_2|\!)$ is obtained from s by replacing occurrences of $select\, t_1\, t_2$ with $select\, x\, t_2$. Hence, $d(s(\!|select\, x\, t_2|\!)) \leq d(s)$, which means that $d(u) > d(u')$.
- The remaining two cases follow in a similar manner. ∎

6.3 *Counterexamples*

In the previous section, we have shown that we can safely execute *Full Async RaTT* programs by first transforming them into corresponding *Async RaTT* programs of the same type and then executing the resulting program on the machine presented in section 4. In this section, we show that this transformation is crucial since we cannot execute *Full Async RaTT* programs directly even with a suitable extension of the operational semantics.

Full Async RaTT removes the restriction of *Async RaTT* that *adv* and *select* can only be used on values. Therefore, if applied to *Full Async RaTT* terms directly, the operational semantics of section 4 would fail in a trivial manner by virtue of not evaluating the argument of *adv* and *select*. We address this by generalising the semantics in a straightforward way:

$$\frac{\langle t; \eta_N \rangle \Downarrow^\iota \left\langle wait_\kappa; \eta'_N \right\rangle}{\langle adv\, t; \eta_N\, \langle \kappa \mapsto v \rangle\, \eta_L \rangle \Downarrow^\iota \left\langle v; \eta'_N\, \langle \kappa \mapsto v \rangle\, \eta_L \right\rangle}$$

$$\frac{\langle t; \eta_N \rangle \Downarrow^\iota \left\langle l; \eta'_N \right\rangle \qquad \left\langle \eta'_N(l); \eta'_N\, \langle \kappa \mapsto v \rangle\, \eta_L \right\rangle \Downarrow^\iota \langle w; \sigma \rangle}{\langle adv\, t; \eta_N\, \langle \kappa \mapsto v \rangle\, \eta_L \rangle \Downarrow^\iota \langle w; \sigma \rangle}$$

Instead of expecting the argument of *adv* to be a value, we first evaluate the argument t to a value and then proceed as in the original semantics, but with a potentially updated now heap

η'_N. A similar generalisation can be made for *select*. We also have to generalise the semantics for *delay*:

$$\frac{\langle \theta; \sigma \rangle \Downarrow^\iota \langle \Theta; \sigma' \rangle \qquad l = alloc^\Theta(\sigma')}{\left\langle delay_\theta\, t; \sigma \right\rangle \Downarrow^\iota \langle l; (\sigma', l \mapsto t) \rangle}$$

In this generalisation, the evaluation of the clock expression θ to a clock Θ is no longer defined by a simple evaluation function $|\cdot|$. Instead, because θ may contain terms that have to be evaluated, the evaluation of θ must be performed on the machine and may result in a new store σ'. We elide the definition of clock expression evaluation $\langle \theta; \sigma \rangle \Downarrow^\iota \langle \Theta; \sigma' \rangle$, as it is straightforward.

In the following, we construct two examples that demonstrate that this semantics may get stuck as it tries to dereference a delayed computation that has been garbage collected in a previous computation step. For the first example, consider the following function, adapted from Bahr (2022), that takes a signal of numbers and produces a new signal that simply repeats every other element of the input signal:

$$\begin{aligned}
&stutter : Sig\ Nat \to Sig\ Nat\\
&stutter\ (x :: xs) = x :: delay\ (x :: delay\ (stutter\ (adv\ (tail\ (adv\ xs)))))
\end{aligned}$$

Note that *stutter* is one of the rare examples that requires multiple ticks in the typing context in order to type check. This definition elaborates into the following *Full Async RaTT* term:

$$\begin{aligned}
stutter = fix\, r.\lambda s.let\, x = \pi_1\ (out\ s)\ in\ let\, xs = \pi_2\ (out\ s)\ in\\
x :: delay_{cl(xs)}\ (x :: delay_{cl(\pi_2\ (out\ (adv\, xs)))}\ (adv_\forall\ r\ (adv\ (\pi_2\ (out\ (adv\, xs)))))))
\end{aligned}$$

We assume a push-only channel $nat :_p Nat \in \Delta$ so that we can apply *stutter* to the signal $0 :: sigAwait_{nat}$ of type $Sig\ Nat$. Following the naming convention from section 4.3, we use $stutter'$ and $sigAwait'_{nat}$ for the variants of *stutter* and $sigAwait_{nat}$ that use *dfix* instead of *fix*. As in section 4.3, we write the machine's store as just the list of its heap locations, and write the contents of the locations separately underneath. We do not give the clocks of the heap locations explicitly, since all locations in this and the next example have the clock $\{nat\}$.

We start with the initialisation step:

$$\begin{aligned}
&\left\langle stutter\ (0 :: sigAwait_{nat}); \emptyset \right\rangle \overset{x \mapsto 0}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2; \emptyset \rangle\\
\text{where}\quad &l_1 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat}\\
&l_2 \mapsto 0 :: delay_{cl(\pi_2\ (out\ (adv\, l_1)))}\ (adv_\forall\ stutter'\ (adv\ (\pi_2\ (out\ (adv\, l_1)))))
\end{aligned}$$

In the subsequent input steps, we assume that *nat* produces increasing numbers. As expected, the program repeats the output 0 from the initialisation step:

$$\begin{aligned}
\langle x \mapsto l_2; l_1, l_2; \emptyset \rangle &\overset{nat \mapsto 1}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2\ \langle nat \mapsto 1 \rangle\ \emptyset; \emptyset \rangle \overset{x \mapsto 0}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4; \emptyset \rangle\\
\text{where}\quad &l_3 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat}\\
&l_4 \mapsto adv_\forall\ stutter'\ (adv\ (\pi_2\ (out\ (adv\, l_1))))
\end{aligned}$$

Here l_3 is allocated while evaluating the clock expression $cl\,(\pi_2\ (out\ (adv\, l_1)))$ of the inner *delay*. However, as we can already see, the delayed computation stored at l_4 references l_1, which has already been garbage collected. As a result, the machine gets stuck after the next

input step:

$$\langle x \mapsto l_4; l_3, l_4; \emptyset \rangle \overset{nat \mapsto 2}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4\, \langle nat \mapsto 2 \rangle\, \emptyset; \emptyset \rangle \Longrightarrow$$

The underlying issue that makes *stutter* fail is that it contains two nested occurrences of *delay*, which require two ticks in the context to type check. This results in two nested occurrences of *adv* in the term stored at l_4, which the evaluation semantics cannot support due to its aggressive garbage collection.

But even if we do not allow for two nested occurrences of *delay*, we can still construct a (rather contrived) *Full Async RaTT* program, adapted from Bahr et al. (2019), that cannot be directly run by the machine:

$$\begin{aligned}
&\mathit{leaky} : (1 \to 1) \to \mathit{Sig}\ A \to \mathit{Sig}\ A \\
&\mathit{leaky}\ f\ (x :: xs) = x :: \mathit{delay}\ (\mathit{leaky}\ (\lambda\, y.\, \mathit{adv}\ (f\ (); xs); ())\ (\mathit{adv}\ (f\ (); xs)))
\end{aligned}$$

Here we use the notation $s; t$ as shorthand for $\mathit{let}\, x = s\, \mathit{in}\, t$ for some fresh variable x. The definition of *leaky* is quite elaborate, but it simply repeats the input signal it is given. That is, $\mathit{leaky}\, \mathit{id} : \mathit{Sig}\, A \to \mathit{Sig}\, A$ is equivalent to the identity function. The above definition elaborates into the following *Full Async RaTT* term:

$$\begin{aligned}
\mathit{leaky} = \mathit{fix}\, r.\lambda f.\lambda s.\mathit{let}\, x = \pi_1\ (\mathit{out}\ s)\ \mathit{in}\ \mathit{let}\, xs = \pi_2\ (\mathit{out}\ s)\ \mathit{in} \\
x :: \mathit{delay}_{cl(f\ ();xs)}\ (\mathit{adv}_\forall\ r\ (\lambda y.\mathit{adv}\ (f\ (); xs); ())\ (\mathit{adv}\ (f\ (); xs)))
\end{aligned}$$

We again assume a push-only channel $\mathit{nat} :_p \mathit{Nat} \in \Delta$ so that we can construct the program $\mathit{leaky}\, \mathit{id}\, (0 :: \mathit{sigAwait}_{nat})$ of type $\mathit{Sig}\, \mathit{Nat}$. We start with the initialisation step:

$$\begin{aligned}
&\left\langle \mathit{leaky}\, \mathit{id}\, (0 :: \mathit{sigAwait}_{nat}); \emptyset \right\rangle \overset{x \mapsto 0}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2; \emptyset \rangle \\
&\quad \text{where} \quad l_1 \mapsto \mathit{adv}\, \mathit{wait}_{nat} :: \mathit{adv}_\forall\, \mathit{sigAwait}'_{nat} \\
&\qquad\qquad l_2 \mapsto \mathit{adv}_\forall\, \mathit{leaky}'\ (\lambda y.\mathit{adv}\ (\mathit{id}\ (); l_1); ())\ (\mathit{adv}\ (\mathit{id}\ (); l_1))
\end{aligned}$$

Given an input 1 on the *nat* channel, the program simply repeats this input as expected:

$$\begin{aligned}
&\langle x \mapsto l_2; l_1, l_2; \emptyset \rangle \overset{nat \mapsto 1}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2\, \langle nat \mapsto 1 \rangle\, \emptyset; \emptyset \rangle \overset{x \mapsto 1}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4; \emptyset \rangle \\
&\quad \text{where} \quad l_3 \mapsto \mathit{adv}\, \mathit{wait}_{nat} :: \mathit{adv}_\forall\, \mathit{sigAwait}'_{nat} \\
&\qquad\qquad l_4 \mapsto \mathit{adv}_\forall\, \mathit{leaky}'\ (\lambda y.\mathit{adv}\ ((\lambda y.\mathit{adv}\ (\mathit{id}\ (); l_1); ())\ (); l_3); ()) \\
&\qquad\qquad\qquad\qquad (\mathit{adv}\ ((\lambda y.\mathit{adv}\ (\mathit{id}\ (); l_1); ())\ (); l_3))
\end{aligned}$$

After the next input transition, the machine gets stuck trying to evaluate the subterm $(\lambda y.\mathit{adv}\ (\mathit{id}\ (); l_1); ())$ that it encounters at heap location l_4. Evaluation of this term requires dereferencing the heap location l_1, which has been garbage collected by this time:

$$\langle x \mapsto l_4; l_3, l_4; \emptyset \rangle \overset{nat \mapsto 2}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4\, \langle nat \mapsto 2 \rangle\, \emptyset; \emptyset \rangle \Longrightarrow$$

One might try to solve this problem by holding on to heap locations for a bit longer. However, the result would be that we could run this program for a bit longer, but it would eventually get stuck again since the reference to l_1 will remain in the store indefinitely. In other words, it would require an unbounded store to safely run this program directly.

For comparison, we consider the following *Async RaTT* program $leaky_T$ that we obtain after applying the program transformation to *leaky*:

$$\begin{aligned} leaky_T = fix\, r.\lambda f.\lambda s.let\, x = \pi_1\ (out\ s)\ in\ let\ xs = \pi_2\ (out\ s)\ in \\ x :: let\ d = f\ (); xs\ in\ delay_{cl(d)}\ (adv_\forall\ r\ (\lambda y.adv\ d; ())\ (adv\ d)) \end{aligned}$$

Let us try to run this program on the machine with the same inputs from *nat*:

$$\begin{aligned} &\left\langle leaky_T\ id\ (0 :: sigAwait_{nat}); \emptyset \right\rangle \overset{x \mapsto 0}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2; \emptyset \rangle \\ &\quad \text{where} \quad l_1 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat} \\ &\qquad\qquad l_2 \mapsto adv_\forall\ leaky'_T\ (\lambda y.adv\ l_1; ())\ (adv\ l_1) \end{aligned}$$

As we can see, *adv* is now directly applied to the heap location l_1 instead of a term that will evaluate to l_1. This avoids the problem that an unevaluated term may contain references to heap locations that will be garbage collected before the machine has the chance to evaluate the term. We can see this after the next output transition:

$$\begin{aligned} \langle x \mapsto l_2; l_1, l_2; \emptyset \rangle &\overset{nat \mapsto 1}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2\ \langle nat \mapsto 1 \rangle\ \emptyset; \emptyset \rangle \overset{x \mapsto 1}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4; \emptyset \rangle \\ &\text{where} \quad l_3 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat} \\ &\qquad\quad l_4 \mapsto adv_\forall\ leaky'_T\ (\lambda y.adv\ l_3; ())\ (adv\ l_3) \end{aligned}$$

Unlike the untransformed program, the heap location l_4 no longer contains a reference to the garbage-collected heap location l_1. We can thus safely continue the execution of the program:

$$\begin{aligned} \langle x \mapsto l_4; l_3, l_4; \emptyset \rangle &\overset{nat \mapsto 2}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4\ \langle nat \mapsto 2 \rangle\ \emptyset; \emptyset \rangle \overset{x \mapsto 2}{\Longrightarrow} \langle x \mapsto l_6; l_5, l_6; \emptyset \rangle \\ &\text{where} \quad l_5 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat} \\ &\qquad\quad l_6 \mapsto adv_\forall\ leaky'_T\ (\lambda y.adv\ l_5; ())\ (adv\ l_5) \end{aligned}$$

7 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 such as the lack of space leaks proved here. 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, approach of using an aggressive garbage collection technique for avoiding space leaks is due to Krishnaswami (2013). Krishnaswami's calculus uses 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 ◯ 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 later modality ∋ as a delay on a set of input channels. From the perspective of the model, ∋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 λ_{widget} defined by Graulund et al. (2021), which takes $\diamond$ as a type constructor primitive and endows it with a synchronisation primitive similar to *select* in *Async RaTT*. However, the programming primitives in λ_{widget} are very different from the ones used 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 $\exists$ is not expressible: Unlike $\exists A$, 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 to 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 $\mathit{Sig}\ (\mathit{Time} \to A)$, i.e., at each tick of the clock, we get a new function $\mathit{Time} \to A$ that describes the time-varying value of the signal until the next tick of the clock (Faurby Klausen et al. 2026).

Futures, first implemented in MultiLisp (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 $\mathit{Future}\ A$ 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 $\mathit{Future}\ \mathit{Buffer}$ instead of blocking the caller until the file was read into a buffer. *Async RaTT* can provide a similar interface using the type modality $\exists$, either directly or by defining Future as a guarded recursive type $\mathit{Future}\ A \cong A + \exists(\mathit{Future}\ A)$ 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 $\mathit{Future}\ \mathit{Buffer}$ simply as a family of channels $\mathit{readFile}_f :_p \mathit{Buffer}$. 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) use 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.

8 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 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 our results could be reproduced in a dual-context language. Even though Fitch-style typing rules use non-standard operations on contexts, *Full Async RaTT* has been implemented as an embedded language in Haskell (Bahr et al. 2024a), thus giving programmers access to a combination of features from *Full Async RaTT* and the host programming language. This has been further demonstrated by implementations of a GUI library (Disch et al. 2025), a property-based testing library (Nielsen et al. 2026), and a push-pull FRP library (Faurby Klausen et al. 2026).

Acknowledgements

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

Conflicts of interest

The authors report no conflict of interest.

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 (2001), 657–683. doi:10.1145/504709.504712

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

Patrick Bahr, 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. doi:10.1145/3341713

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

Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam. 2024a. Asynchronous Reactive Programming with Modal Types in Haskell. In *Practical Aspects of Declarative Languages*. 18–36. doi:10.1007/978-3-031-52038-9_2

Patrick Bahr, Emil Houlborg, and Gregers Thomas Skat Rørdam. 2024b. AsyncRattus Haskell package. https://hackage.haskell.org/package/AsyncRattus

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

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

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. doi:10.1017/S0960129519000197

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

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*. 178–188. doi: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*. 361–372. doi:10.1145/2535838.2535881

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*. 258–275. doi:10.1007/978-3-319-89366-2_14

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM* 48, 3 (2001), 555–604. doi:10.1145/382780.382785

Jean-Claude Disch, Asger Heegaard, and Patrick Bahr. 2025. Functional Reactive GUI Programming with Modal Types. In *Trends in Functional Programming*. 93–114. doi:10.1007/978-3-031-99751-8_5

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. 263–273. doi:10.1145/258948.258973

Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. 25–36. doi:10.1145/1596638.1596643

Lasse Faurby Klausen, Philip Kristian Møller Flyvholm, and Patrick Bahr. 2026. Push-Pull Modal Functional Reactive Programming. (2026). Symposium on Trends in Functional Programming, to appear.

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

Robert H. Halstead. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (1985), 501–538. doi: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*. 49–60. doi: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)*. 54:1–54:9. doi: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. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII). doi:10.1016/j.entcs.2012.08.015

Eugen Kiss. 2014. 7GUIs: A GUI Programming Benchmark. https://eugenkiss.github.io/7guis/tasks

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

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

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 45–58. doi: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)*. 255–266. doi:10.1109/LICS.2000.855774

Christian Emil Nielsen, Mathias Faber Kristiansen, and Patrick Bahr. 2026. Property-Based Testing for Asynchronous Functional Reactive Programming Using Linear Temporal Logic. In *Practical Aspects of Declarative Languages*. 39–56. doi:10.1007/978-3-032-15981-6_3

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

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

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

Marc Pouzet. 2006. Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI. https://hal.science/hal-03090137v1

John H. Reppy. 1999. *Concurrent programming in ML*. Cambridge University Press, Cambridge, England.

Terese. 2003. *Term Rewriting Systems*. Cambridge University Press.

Vincent van Oostrom. 1997. Developing developments. *Theoretical Computer Science* 175, 1 (1997), 159–181. doi:10.1016/S0304-3975(96)00173-9

Femke van Raamsdonk. 1996. *Confluence and Normalisation of Higher-Order Rewriting*. PhD Dissertation. Vrije Universiteit Amsterdam.

A Proof of the fundamental property

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

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

Lemma A.1.

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

Proof By a straightforward case analysis. ∎

Lemma A.2.

(i) $[tick_{\kappa \mapsto v}\,(\eta)]_{\Theta} = tick_{\kappa \mapsto v}\,([\eta]_{\Theta})$.
(ii) $[gc(\sigma)]_{\Theta} = gc([\sigma]_{\Theta})$.

Proof By a straightforward case analysis. ∎

Lemma A.3. *Let* $\gamma \in C_{\Delta}[\![\Gamma, \Gamma']\!]\,(n, \sigma)$ *such that* Γ' *is tick-free. Then* $\gamma|_{\Gamma} \in C_{\Delta}[\![\Gamma]\!]\,(n, \sigma)$.

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

Lemma A.4. *If* $\gamma \in C_{\Delta}[\![\Gamma]\!]\,(n, \sigma)$, *then* $\gamma|_{\Gamma^{\Box}} \in C_{\Delta}[\![\Gamma^{\Box}]\!]\,(n, \emptyset)$.

Proof By a straightforward induction on the length of Γ using Lemma 5.3 and Lemma 5.5. ∎

Lemma A.5. *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 $\vdash \iota : \Delta$. 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.6. *If* $\Gamma \vdash_{\Delta} \theta : Clock$ *and* $\gamma \in C_{\Delta}[\![\Gamma]\!]\,(w)$, *then* $\theta\gamma$ *is a closed clock expression and* $|\theta\gamma| \subseteq dom_p\,(\Delta)$.

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

- $$\frac{\Gamma \vdash_{\Delta} \theta : Clock \qquad \Gamma \vdash_{\Delta} \theta' : Clock}{\Gamma \vdash_{\Delta} \theta \sqcup \theta' : Clock}$$
  By induction, $\theta\gamma$ and $\theta'\gamma$ are closed and $|\theta\gamma| \subseteq dom_p\,(\Delta)$. Hence, $(\theta \sqcup \theta')\gamma = \theta\gamma \sqcup \theta'\gamma$ is closed and $|(\theta \sqcup \theta')\gamma| = |\theta\gamma| \cup |\theta'\gamma| \subseteq dom_p\,(\Delta)$.
- $$\frac{\Gamma \vdash_{\Delta} v : \circledcirc A}{\Gamma \vdash_{\Delta} cl(v) : Clock}$$
  $\Gamma \vdash_{\Delta} v : \circledcirc A$ implies that $v\gamma \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!]\,(w)$ (because either v is a variable or $v = wait_{\kappa}$ for some channel κ). Hence, $v\gamma \in Loc_{\Delta} \cup \left\{ wait_{\kappa} \,\middle|\, \kappa \in dom_p\,(\Delta) \right\}$ and thus $cl(v)\,\gamma$ is a closed clock expression and $|cl(v)\,\gamma| \subseteq dom_p\,(\Delta)$.

∎

Lemma A.7. *Let* $\eta_N \in Heap^{\kappa}$, $v \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!]\,\big(n+1, (\eta_N, [\eta_L]_{\kappa \notin})\big)$, $\vdash \iota : \Delta$, $\vdash w : \Delta(\kappa)$ *and* $\kappa \in |cl(v)|$. *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* $\langle adv\,v; \sigma \rangle \Downarrow^{\iota} \langle v'; \sigma' \rangle$.

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

- In the former case, we have by the definition of the value relation and Lemma A.2,

$$(\eta_N, [\eta_L]_{\kappa\notin})(l) \in \mathcal{T}_\Delta[\![A]\!]\left(n, \left[\eta_N\,\langle\kappa \mapsto w\rangle\,[\eta_L]_{\kappa\notin}\right]_{cl(l)}\right).$$

  In turn, this implies by Lemma 5.3 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 $\kappa \in cl(l)$ 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 $\langle adv\, v; \sigma\rangle \Downarrow^\iota \langle v'; \sigma'\rangle$.
- In the latter case, we know that $\kappa' = \kappa$ because $\kappa \in |cl(wait_{\kappa'})| = \{\kappa'\}$. Moreover, we have that $\kappa :_c A \in \Delta$ for $c \in \{p, bp\}$ 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 $\langle adv\, wait_\kappa; \sigma\rangle \Downarrow^\iota \langle w; \sigma\rangle$. Moreover, by Lemma 5.9, $w \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$. ∎

We proceed with two lemmas that allow us to reason about contexts of the form $\Gamma^\bigcirc$, defined in section 5.1, which we need in order to apply Lemma 5.2.

Lemma A.8. *If $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma)$, then $\gamma|_{\Gamma^\bigcirc} \in C_\Delta[\![\Gamma^\bigcirc]\!]\,(n, \sigma)$.*

Proof By a straightforward induction on the length of Γ and using Lemma A.4 and Lemma 5.3. ∎

Lemma A.9. *If $\Gamma, \Gamma' \vdash_\Delta \theta : Clock$, then $\Gamma^\bigcirc, \Gamma' \vdash_\Delta \theta : Clock$.*

Proof Straightforward induction on $\Gamma, \Gamma' \vdash_\Delta \theta : Clock$. ∎

We can now give the full proof of the fundamental property:

Theorem 5.8 (Fundamental property)**.** *If $\Gamma \vdash_\Delta$, $\Gamma \vdash_\Delta t : A$, and $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma)$, then $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma)$.*

Proof We proceed by induction on the size of t. If $t\gamma$ is a value, it suffices to show that $t\gamma \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$, according to Lemma 5.4. In all other cases, to prove $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma)$, we assume some input buffer $\vdash \iota : \Delta$ and store $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$, and show that there exists σ'' and v such that $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ and $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$. By Lemma 5.3 we may assume that $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma')$.

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

  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.3. If Γ' is not tick-free, then A is stable. Hence, $x : A \in \Gamma^\square$ and thus $x \in dom\,(\gamma|_{\Gamma^\square})$. By Lemma A.4, $\gamma|_{\Gamma^\square} \in C_\Delta[\![\Gamma^\square]\!]\,(n, \emptyset)$. Since $\Gamma^\square$ is tick-free we thus have $x\gamma \in \mathcal{V}_\Delta[\![A]\!]\,(n, \emptyset)$ by Lemma A.3. Hence, by Lemma 5.3, we have that $x\gamma \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$.

- $$\frac{}{\Gamma \vdash_\Delta () : 1}$$

  Follows immediately by definition.

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

  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 5.3 and Lemma 5.1, $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma'')$ and thus

  $$\gamma[x \mapsto v] \in C_\Delta[\![\Gamma, x : A]\!]\,(n, \sigma'')\,.$$

  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 $\langle (\mathit{let}\, x = s\, \mathit{in}\, t)\gamma; \sigma'\rangle \Downarrow^\iota \langle w; \sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!]\,(n, \sigma''')$.

- $\dfrac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda x.t : A \to B}$

  We show that $\lambda x.t\gamma \in \mathcal{V}_\Delta[\![A \to B]\!]\,(n, \sigma)$. To this end, we assume $w \gtrsim^\Delta (n, \sigma)$ and $v \in \mathcal{V}_\Delta[\![A]\!]\,(w)$, with the goal of showing $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(w)$. By Lemma 5.3, $\gamma \in C_\Delta[\![\Gamma]\!]\,(w)$. Thus, by definition, $\gamma[x \mapsto v] \in C_\Delta[\![\Gamma, x : A]\!]\,(w)$. By induction, we then have that $t\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!]\,(w)$. Since all elements in the range of γ are closed terms, $t\gamma[x \mapsto v] = (t\gamma)[v/x]$ and thus $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(w)$.

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

  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 5.1, $\sigma'' \sqsupseteq^\Delta_\checkmark \sigma$, 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 $\sigma''' \sqsupseteq^\Delta_\checkmark gc(\sigma'')$ by Lemma A.1 and Lemma 5.1. 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 $\langle (s\gamma)(t\gamma); \sigma'\rangle \Downarrow^\iota \langle w; \sigma''''\rangle$.

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

  By induction, we have $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma)$, which means that $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$. We also have $t'\gamma \in \mathcal{T}_\Delta[\![B]\!]\,(n, \sigma)$ by induction, which by Lemma 5.1 means that $\langle t'\gamma; \sigma''\rangle \Downarrow^\iota \langle v'; \sigma'''\rangle$ for some $v' \in \mathcal{V}_\Delta[\![B]\!]\,(n, \sigma''')$. Hence, $\langle (t\gamma, t'\gamma)\,; \sigma'\rangle \Downarrow^\iota \langle (v, v')\,; \sigma'''\rangle$, and by Lemma 5.3, $(v, v') \in \mathcal{V}_\Delta[\![A \times B]\!]\,(n, \sigma''')$.

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

  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, $\langle \pi_i\, t\gamma; \sigma'\rangle \Downarrow^\iota \langle v_i; \sigma''\rangle$.

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

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, $\langle in_i\, t\gamma;\sigma'\rangle \Downarrow^\iota \langle in_i\, v;\sigma''\rangle$ and $in_i\, v \in \mathcal{V}_\Delta[\![A_1 + A_2]\!]\,(n,\sigma'')$.

- $\dfrac{\Gamma, x : A_i \vdash_\Delta t_i : B \qquad \Gamma \vdash_\Delta t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash_\Delta \mathit{case}\, t\, \mathit{of}\, in_1\, x.t_1; in_2\, x.t_2 : B}$

By induction, we have $t\gamma \in \mathcal{T}_\Delta[\![A_1 + A_2]\!]\,(n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle in_i\, v;\sigma''\rangle$ for some $i \in \{1,2\}$ such that $v \in \mathcal{V}_\Delta[\![A_i]\!]\,(n,\sigma'')$. By Lemma 5.3 and Lemma 5.1, $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma'')$ and thus $\gamma[x \mapsto v] \in \mathcal{C}_\Delta[\![\Gamma, x : A_i]\!]\,(n,\sigma'')$. 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 $\langle (\mathit{case}\, t\, \mathit{of}\, in_1\, x.t_1; in_2\, x.t_2)\gamma;\sigma'\rangle \Downarrow^\iota \langle w;\sigma'''\rangle$, as well.

- $\dfrac{\Gamma, \checkmark_\theta \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta \theta : \mathit{Clock}}{\Gamma \vdash_\Delta \mathit{delay}_\theta\, t : \triangleleft A}$

By definition of the machine we have that $\langle \mathit{delay}_{\theta\gamma}\, t\gamma;\sigma'\rangle \Downarrow^\iota \langle l;\sigma''\rangle$, where $\sigma'' = \sigma', l \mapsto t\gamma$ and $cl(l) = |\theta\gamma|$. By Lemma 5.2, we have that $\Gamma^\bigcirc, \checkmark_\theta \vdash_\Delta t : A$. Let $\gamma' = \gamma|_{\Gamma^\bigcirc}$. By Lemma A.9, we have that $\Gamma^\bigcirc \vdash_\Delta \theta : \mathit{Clock}$. Consequently, $\theta\gamma' = \theta\gamma$. By Lemma A.6, $|\theta\gamma| = |\theta\gamma'| \subseteq \mathit{dom}_p\,(\Delta)$. It remains to be shown that $l \in \mathcal{V}_\Delta[\![\triangleleft A]\!]\,(n,\sigma'')$. For the case where $n = 0$, this follows immediately from the fact that $|\theta\gamma| \subseteq \mathit{dom}_p\,(\Delta)$.

Assume that $n = n' + 1$, $\kappa \in \Theta$, and $\vdash v : \Delta(\kappa)$, where $\Theta = |\theta\gamma|$. By Lemma 5.3, we have that $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n'+1,\sigma'')$. We then have $\gamma' \in \mathcal{C}_\Delta[\![\Gamma^\bigcirc]\!]\,(n'+1,\sigma'')$ by Lemma A.8, and in turn we have $\gamma' \in \mathcal{C}_\Delta[\![\Gamma^\bigcirc]\!]\,(n'+1, gc(\sigma''))$ by Lemma 5.6. By definition, we thus have that

$$\begin{aligned}\gamma' &\in \mathcal{C}_\Delta[\![\Gamma^\bigcirc, \checkmark_\theta]\!]\,\big(n', [gc(\sigma'')]_{\kappa\in}\,\langle\kappa \mapsto v\rangle\,[gc(\sigma'')]_{\kappa\notin}\big)\\ &= \mathcal{C}_\Delta[\![\Gamma^\bigcirc, \checkmark_\theta]\!]\,(n', \mathit{tick}_{\kappa\mapsto v}\,(gc(\sigma''))),\end{aligned}$$

and thus $\gamma' \in \mathcal{C}_\Delta[\![\Gamma^\bigcirc, \checkmark_\theta]\!]\,\big(n', [\mathit{tick}_{\kappa\mapsto v}\,(gc(\sigma''))]_\Theta\big)$ according to Lemma 5.7. Hence, we can apply the induction hypothesis to conclude that

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

Since $\Gamma^\bigcirc, \checkmark_\theta \vdash_\Delta t : A$, we know that $t\gamma = t\gamma'$. Because $\sigma''(l) = t\gamma = t\gamma'$, we thus have that $l \in \mathcal{V}_\Delta[\![\triangleleft A]\!]\,(n,\sigma'')$.

- $\dfrac{}{\Gamma \vdash_\Delta \mathit{never} : \triangleleft A}$

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

- $\dfrac{\kappa :_c A \in \Delta, c \in \{p, bp\}}{\Gamma \vdash_\Delta \mathit{wait}_\kappa : \triangleleft A}$

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

$$\frac{\kappa :_c A \in \Delta, c \in \{b, bp\}}{\Gamma \vdash_\Delta read_\kappa : A}$$

Since $\vdash \iota : \Delta$, we know that $\vdash \iota(\kappa) : A$. Hence, by definition of the machine $\langle read_\kappa ; \sigma' \rangle \Downarrow^\iota \langle \iota(\kappa); \sigma' \rangle$. Moreover, by Lemma 5.9 $\iota(\kappa) \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma')$.

$$\frac{\Gamma \vdash_\Delta v : \circledcirc A \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_{cl(v)}, \Gamma' \vdash_\Delta adv\, v : A}$$

By Lemma 5.3 we have that $\gamma \in C_\Delta[\![\Gamma, \checkmark_{cl(v)}, \Gamma']\!]\,(n, \sigma')$ and by Lemma A.3, $\gamma|_\Gamma \in C_\Delta[\![\Gamma, \checkmark_{cl(v)}]\!]\,(n, \sigma')$. Let $\Theta = |cl(v)\,\gamma|_\Gamma|$. By definition of the context relation, Θ is well-defined and a subset of $dom_p\,(\Delta)$. By definition of the context relation we also find $\kappa \in \Theta$, $\eta_N, \eta_L, \eta'_N, \eta'_L$ such that $\sigma' = \eta_N \langle \kappa \mapsto w \rangle \eta_L$, $\vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = [\eta'_N]_\Theta$, $[\eta_L]_\Theta = [\eta'_L]_\Theta$, and $\gamma|_\Gamma \in C_\Delta[\![\Gamma]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$. By induction, we thus have that $v\gamma \in \mathcal{T}_\Delta[\![\circledcirc A]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$. Since $v\gamma$ is a value we have $v\gamma \in \mathcal{V}_\Delta[\![\circledcirc A]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$ by Lemma A.5. By Lemma 5.7 we then have that

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

By Lemma A.7, we then find a reduction $\langle adv\, v\gamma; \sigma' \rangle \Downarrow^\iota \langle v'; \sigma'' \rangle$ with $v' \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$.

$$\frac{\Gamma \vdash_\Delta v_1 : \circledcirc A_1 \qquad \Gamma \vdash_\Delta v_2 : \circledcirc A_2 \qquad \theta = cl(v_1) \sqcup cl(v_2) \qquad \Gamma' \text{ tick-free}}{\Gamma, \checkmark_\theta, \Gamma' \vdash_\Delta select\, v_1\, v_2 : ((A_1 \times \circledcirc A_2) + (\circledcirc A_1 \times A_2)) + (A_1 \times A_2)}$$

By Lemma 5.3 we have that $\gamma \in C_\Delta[\![\Gamma, \checkmark_\theta, \Gamma']\!]\,(n, \sigma')$ and by Lemma A.3, $\gamma|_\Gamma \in C_\Delta[\![\Gamma, \checkmark_\theta]\!]\,(n, \sigma')$. Let $\Theta_1 = |cl(v_1)\,\gamma|_\Gamma|$, $\Theta_2 = |cl(v_2)\,\gamma|_\Gamma|$, and $\Theta = \Theta_1 \cup \Theta_2$. According to the definition of the context relation, Θ is well-defined and a subset of $dom_p\,(\Delta)$. By definition of the context relation we also find $\kappa \in \Theta$, $\eta_N, \eta_L, \eta'_N, \eta'_L$ such that $\sigma' = \eta_N \langle \kappa \mapsto w \rangle \eta_L$, $\vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = [\eta'_N]_\Theta$, $[\eta_L]_\Theta = [\eta'_L]_\Theta$, and $\gamma|_\Gamma \in C_\Delta[\![\Gamma]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$. By induction hypothesis, we thus have that $v_i\gamma \in \mathcal{T}_\Delta[\![\circledcirc A_i]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$ for all $i \in \{1,2\}$. Since $v_i\gamma$ are values, we also have that $v_i\gamma \in \mathcal{V}_\Delta[\![\circledcirc A_i]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$ by Lemma A.5. By Lemma 5.7 we then have that $v_i\gamma \in \mathcal{V}_\Delta[\![\circledcirc A_i]\!]\left(n+1, (\eta_N, [\eta_L]_{\kappa\notin})\right)$ 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.7, there is a reduction $\langle adv\, v_i\gamma; \sigma' \rangle \Downarrow^\iota \langle u_i; \sigma'' \rangle$ with $u_i \in \mathcal{V}_\Delta[\![A_i]\!]\,(n, \sigma'')$. Therefore, by definition of the evaluation semantics, we have the reduction $\langle select\, v_1\gamma\, v_2\gamma; \sigma' \rangle \Downarrow^\iota \langle in_1(in_i\,(u_1, u_2)); \sigma'' \rangle$ with $u_j = v_j\gamma$. It thus remains to be shown that $v_j\gamma \in \mathcal{V}_\Delta[\![\circledcirc A_j]\!]\,(n, \sigma'')$. There are two cases to consider.
  - Let $v_j\gamma = l$ for some $l \in Loc_\Delta$. From $l \in \mathcal{V}_\Delta[\![\circledcirc A_j]\!]\left(n+1, (\eta_N, [\eta_L]_{\kappa\notin})\right)$ and the fact that $\kappa \notin \Theta_j$, we obtain that $l \in \mathcal{V}_\Delta[\![\circledcirc A_j]\!]\left(n+1, [\eta_L]_{\kappa\notin}\right)$ by using Lemma 5.7. In particular, we use the fact that $[\eta_N]_{\Theta_j} = \emptyset$ since $\eta_N \in Heap^\kappa$ and $\kappa \notin \Theta_j$. Since $[\eta_L]_{\kappa\notin} \sqsubseteq^\Delta_\checkmark \sigma'$ and, by Lemma 5.1, $\sigma' \sqsubseteq^\Delta_\checkmark \sigma''$, we can then use Lemma 5.3 to conclude that $l \in \mathcal{V}_\Delta[\![\circledcirc A_j]\!]\,(n, \sigma'')$.

  - Let $v_j\gamma = wait_{\kappa'}$ for some channel κ'. But then $\Gamma \vdash_\Delta v_j : \exists A_j$ is due to $\kappa' :_p A_j \in \Delta$ or $\kappa' :_{bp} A_j \in \Delta$ and thus $v_j\gamma \in \mathcal{V}_\Delta[\![\exists A_j]\!]\,(n, \sigma'')$ follows immediately by definition of the value relation.
- $\kappa \in \Theta_1 \cap \Theta_2$: By Lemma A.7, we obtain a reduction $\langle adv\,(v_1\gamma); \sigma'\rangle \Downarrow^\iota \langle v_1'; \sigma''\rangle$ with $v_1' \in \mathcal{V}_\Delta[\![A_1]\!]\,(n, \sigma'')$, and a reduction $\langle adv\,v_2\gamma; \sigma''\rangle \Downarrow^\iota \langle v_2'; \sigma'''\rangle$ with $v_2' \in \mathcal{V}_\Delta[\![A_2]\!]\,(n, \sigma''')$. By definition, we thus have $\langle select\,(v_1\gamma)\,(v_2\gamma); \sigma'\rangle \Downarrow^\iota \langle in_2\,(v_1', v_2')\,; \sigma'''\rangle$. Moreover, applying Lemma 5.1 and Lemma 5.3, we obtain that $v_1' \in \mathcal{V}_\Delta[\![A_1]\!]\,(n, \sigma''')$, which means that we have $in_2\,(v_1', v_2') \in \mathcal{V}_\Delta[\![((A_1 \times \exists A_2) + (\exists A_1 \times A_2)) + (A_1 \times A_2)]\!]\,(n, \sigma''')$.

- $\dfrac{}{\Gamma \vdash_\Delta 0 : Nat}$

  $0\gamma \in \mathcal{V}_\Delta[\![Nat]\!]\,(n, \sigma)$ follows immediately by definition.
- $\dfrac{\Gamma \vdash_\Delta t : Nat}{\Gamma \vdash_\Delta suc\,t : Nat}$

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

  We claim that the following holds:

$$rec_{Nat}(s\gamma, x\,y.t\gamma, suc^k\,0) \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma) \text{ for all } k \in \mathbb{N} \tag{7}$$

  To show that $rec_{Nat}(s\gamma, x\,y.t\gamma, u\gamma) \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma)$, assume some $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\vdash \iota : \Delta$. By induction hypothesis $u\gamma \in \mathcal{T}_\Delta[\![Nat]\!]\,(n, \sigma)$, which means that $\langle u\gamma; \sigma'\rangle \Downarrow^\iota \langle suc^m\,0; \sigma''\rangle$. By (7) and Lemma 5.1 we have that $\langle rec_{Nat}(s\gamma, x\,y.t\gamma, suc^m\,0); \sigma''\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma''')$. By definition of the machine, we also have that $\langle rec_{Nat}(s\gamma, x\,y.t\gamma, u\gamma); \sigma'\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$.

  We conclude by showing (7) by induction on k.
  - Case $k = 0$: Let $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\vdash \iota : \Delta$. 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, $\langle rec_{Nat}(s\gamma, x\,y.t\gamma, 0); \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ follows.
  - Case $k = l + 1$. Let $\sigma' \sqsupseteq^\Delta_\checkmark \sigma$ and $\vdash \iota : \Delta$. By definition, $\langle suc^k\,0; \sigma'\rangle \Downarrow^\iota \langle suc(suc^l\,0); \sigma'\rangle$. By the induction hypothesis (from the inner induction on k), we have that $\langle rec_{Nat}(s\gamma, x\,y.t\gamma, suc^l\,0); \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$. By Lemma 5.1 and Lemma 5.3, we have $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma'')$ and thus

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

    By induction, we thus obtain that

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

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

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

$$\frac{\Gamma^{\square}, x : \textcircled{\triangledown} A \vdash_{\Delta} t : A}{\Gamma \vdash_{\Delta} \mathit{fix}\, x.t : A}$$

We will show that

$$\mathit{dfix}\, x.t\gamma \in \mathcal{V}_{\Delta}[\![\textcircled{\triangledown} A]\!]\,(m, \emptyset) \text{ for all } m \leq n. \tag{8}$$

Using (8), Lemma A.4, and Lemma 5.3, we then obtain that $(\gamma|_{\Gamma^{\square}})[x \mapsto \mathit{dfix}\, x.t\gamma] \in C_{\Delta}[\![\Gamma^{\square}, x : \textcircled{\triangledown} A]\!]\,(n, \sigma)$. Hence, by the induction hypothesis, we have that

$$t[\mathit{dfix}\, x.t/x]\gamma = t(\gamma|_{\Gamma^{\square}})[x \mapsto \mathit{dfix}\, x.t\gamma] \in \mathcal{T}_{\Delta}[\![A]\!]\,(n, \sigma)$$

which means that we find $\langle t[\mathit{dfix}\, x.t/x]\gamma; \sigma'\rangle \Downarrow^{\iota} \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_{\Delta}[\![A]\!]\,(n, \sigma'')$. By definition of the machine we thus obtain the desired $\langle \mathit{fix}\, x.t\gamma; \sigma'\rangle \Downarrow^{\iota} \langle v; \sigma''\rangle$.
We prove (8) by induction on m.
If $m = 0$, then (8) follows immediately from the fact that $\mathit{dfix}\, x.t\gamma$ is a closed term.
Let $m = m' + 1$. By Lemma 5.3 and Lemma A.4, $\gamma|_{\Gamma^{\square}} \in C_{\Delta}[\![\Gamma^{\square}]\!]\,(m', \emptyset)$. By the induction hypothesis (on (8)) we have $\mathit{dfix}\, x.t\gamma \in \mathcal{V}_{\Delta}[\![\textcircled{\triangledown} A]\!]\,(m', \emptyset)$ and thus

$$(\gamma|_{\Gamma^{\square}})[x \mapsto \mathit{dfix}\, x.t\gamma] \in C_{\Delta}[\![\Gamma^{\square}, x : \textcircled{\triangledown} A]\!]\,(m', \emptyset)\,.$$

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

$$\frac{\Gamma \vdash_{\Delta} v : \textcircled{\triangledown} A}{\Gamma, \checkmark_{\theta}, \Gamma' \vdash_{\Delta} \mathit{adv}_{\triangledown}\, v : A}$$

Since there are no introduction forms for $\textcircled{\triangledown} A$, v must be some variable x. By Lemma 5.3 we have that $\gamma \in C_{\Delta}[\![\Gamma, \checkmark_{\theta}, \Gamma']\!]\,(n, \sigma')$ and by Lemma A.3, $\gamma|_{\Gamma} \in C_{\Delta}[\![\Gamma, \checkmark_{\theta}]\!]\,(n, \sigma')$. Let $\Theta = |\theta\gamma|_{\Gamma}|$. According to 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 $\gamma|_{\Gamma} \in C_{\Delta}[\![\Gamma]\!]\left(n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa\notin})\right)$. By Lemma A.3, we thus have that $\gamma(x) = \mathit{dfix}\, y.t$ with $\mathit{dfix}\, y.t \in \mathcal{V}_{\Delta}[\![\textcircled{\triangledown} A]\!]\left(n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa\notin})\right)$. By definition, this implies that $t[\mathit{dfix}\, y.t/y] \in \mathcal{T}_{\Delta}[\![A]\!]\,(n, \emptyset)$. That is, we find a reduction $\langle t[\mathit{dfix}\, y.t/y]; \sigma'\rangle \Downarrow^{\iota} \langle u; \sigma''\rangle$ with $u \in \mathcal{V}_{\Delta}[\![A]\!]\,(n, \sigma'')$, which by definition means that we also have a reduction $\langle \mathit{adv}_{\triangledown}\, x\gamma; \sigma'\rangle \Downarrow^{\iota} \langle u; \sigma''\rangle$.

$$\frac{\Gamma^{\square} \vdash_{\Delta} t : A}{\Gamma \vdash_{\Delta} \mathit{box}\, t : \square A}$$

We show that $\mathit{box}\, t\gamma \in \mathcal{V}_{\Delta}[\![\square A]\!]\,(n, \sigma)$. By Lemma A.4, $\gamma|_{\Gamma^{\square}} \in C_{\Delta}[\![\Gamma^{\square}]\!]\,(n, \emptyset)$. Hence, by induction, $t\gamma = t\gamma|_{\Gamma^{\square}} \in \mathcal{T}_{\Delta}[\![A]\!]\,(n, \emptyset)$, and thus $\mathit{box}\, t\gamma \in \mathcal{V}_{\Delta}[\![\square A]\!]\,(n, \sigma)$.

$$\frac{\Gamma \vdash_{\Delta} t : \square A}{\Gamma \vdash_{\Delta} \mathit{unbox}\, t : A}$$

By induction hypothesis, we have that $t\gamma \in \mathcal{T}_{\Delta}[\![\square A]\!]\,(n, \sigma)$. That is, $\langle t\gamma; \sigma'\rangle \Downarrow^{\iota} \langle \mathit{box}\, s; \sigma''\rangle$ for some $s \in \mathcal{T}_{\Delta}[\![A]\!]\,(n, \emptyset)$. Hence, $\langle s; \sigma''\rangle \Downarrow^{\iota} \langle v; \sigma'''\rangle$ such that $v \in \mathcal{V}_{\Delta}[\![A]\!]\,(n, \sigma''')$. Moreover, by definition, we have that $\langle \mathit{unbox}\, t\gamma; \sigma'\rangle \Downarrow^{\iota} \langle v; \sigma'''\rangle$.

$$\frac{\Gamma \vdash_{\Delta} t : \mathit{Fix}\, \alpha.A}{\Gamma \vdash_{\Delta} \mathit{out}\, t : A[\textcircled{\exists}(\mathit{Fix}\, \alpha.A)/\alpha]}$$

By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![Fix\,\alpha.A]\!]\,(n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle into\,v;\sigma''\rangle$ for some $v \in \mathcal{V}_\Delta[\![A[(\triangleright)(Fix\,\alpha.A)/\alpha]]\!]\,(n,\sigma'')$. Moreover, by definition of the machine we consequently have $\langle out\,t\gamma;\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle$.

- $\dfrac{\Gamma \vdash_\Delta t : A[(\triangleright)(Fix\,\alpha.A)/\alpha]}{\Gamma \vdash_\Delta into\,t : Fix\,\alpha.A}$

  By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![A[(\triangleright)(Fix\,\alpha.A)/\alpha]]\!]\,(n,\sigma)$, which means that $\langle t\gamma;\sigma'\rangle \Downarrow^\iota \langle v;\sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A[(\triangleright)(Fix\,\alpha.A)/\alpha]]\!]\,(n,\sigma'')$. Hence, by definition, $\langle into\,t\gamma;\sigma'\rangle \Downarrow^\iota \langle into\,v;\sigma''\rangle$ and $into\,v \in \mathcal{V}_\Delta[\![Fix\,\alpha.A]\!]\,(n,\sigma'')$. ■