Asynchronous Modal FRP: Ticking All the Boxes

PATRICK BAHR

IT University of Copenhagen
e-mail: paba@itu.dk

RASMUS EJLERS MØGELBERG

IT University of Copenhagen
e-mail: mogel@itu.dk

Abstract

Over the past fifteen years, a number of languages for functional reactive programming (FRP) have been suggested, which use modal types to ensure properties like causality, productivity and lack of space leaks. So far, almost all of these languages have included a modal operator for delay on a *global* clock. For some applications, however, the notion of 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 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 (Jeffrey, 2014, 2012; Krishnaswami and Benton, 2011; Krishnaswami et al., 2012; Krishnaswami, 2013; Bahr et al., 2019; Bahr, 2022; Jeltsch, 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 $\mathit{Sig}\ A \cong A \times \bigcirc(\mathit{Sig}\ A)$ stating that the current value of the signal is available now, but its future values are only available after the next time step. Using this encoding of signals, one can ensure that all reactive programs are causal. Many modal FRP languages also include a variant of the Nakano (2000) guarded fixed point operator of type $(\bigcirc A \to A) \to A$. The type ensures that recursive calls are only performed in future steps, thus ensuring termination of each step of computation, a property called *productivity*. Often these languages also include a $\Box$ modality used to classify data that is *stable*, in the sense that it can be kept 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 (Jeffrey, 2012; Jeltsch, 2012; Cave et al., 2014; Bahr et al., 2021).

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 $\mathit{Maybe}(\mathit{Char})$. In the modal FRP languages of Krishnaswami (2013); Bahr et al. (2019), 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 et al., 2019; Bahr, 2022; Bahr et al., 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. We refer to such a set θ of input channels as a *clock*. When the signal on an input channel κ is updated, only those output channels whose clock θ contains κ will be updated. For example, the keystroke input channel might be in the clock for the text field content but not the text field colour. Since the program can dynamically change its internal dataflow graph, the clock associated with an output channel may change during execution 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 $\Box$ used to classify stable data, as well as two new modalities: $\textcircled{\exists}$ for asynchronous delays and $\textcircled{\forall}$ for a delay on the global clock. A value of type $\textcircled{\exists}A$ is a pair consisting of a clock θ and a computation that can be executed to return data of type A on the next tick on θ. The type $\textcircled{\exists}A$ can therefore be thought of as an 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.

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

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

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

$$\mathit{switch} : \mathit{Sig}\ A \to \textcircled{\exists}(\mathit{Sig}\ A) \to \mathit{Sig}\ A$$

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.

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

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

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

1.2 Operational semantics and results

We present an operational semantics 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 from the same signal arrives, depending on the kind of the specific input signal. 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 expressivity of *Async RaTT* by developing a small library of signal combinators, along with examples that use the library for GUI programming and

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

Fig. 1. Syntax.

computing integrals and derivatives of signals. The operational semantics is defined in section 4, which also illustrates it with an example, and presents the main results. Section 5 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 limitation 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. This version of the paper is equipped with an appendix detailing 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* 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 present a generalised type system, called *Full Async RaTT*, and show that closed *Full Async RaTT* programs can be transformed into *Async RaTT* programs of the same type.
- 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.
- We include all proofs of the main results: productivity, causality, signal independence, and absence of implicit space leaks.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Typing rules for *Async RaTT*. Rules marked with ⋆ will be generalised in section 6.

2 Async RaTT

In this section, we give an overview of *Async RaTT*, referring to Figures 1 and 2 for the full specification of its syntax and typing rules.

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

$$keyPressed :_{p} Nat, mouseCoord :_{bp} Nat \times Nat, time :_{b} Float \tag{2.1}$$

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

All signals are assumed to have value types, i.e., any declaration $\kappa :_{c} A$ in Δ must have a value type A. Intuitively speaking, value types classify basic values that contain no delayed computations and thus exclude function types and all modal types. The grammar for value types is given in Figure 1.

2.1 Clocks and the later modality ⋵

A clock is effectively a set of push channels (p or bp), that the program may have to react to. For instance, $\emptyset$, $\{keyPressed\}$ and $\{keyPressed, mouseCoord\}$ are all examples of clocks for the example input context given in (2.1) above. The type $⋵A$ is a type of delayed computation on an existentially quantified clock. In other words, a value of type $⋵A$ is a pair of a clock θ and a computation that will produce a value of type A once an update on one of the input channels in θ is received. We refer to such an update as a *tick* on the clock θ. For example, if the associated clock is $\{keyPressed, mouseCoord\}$, then the data of type A can be computed once *keyPressed* or *mouseCoord* receive new input.

Since $⋵A$ are existential types, one can obtain the clock $cl(v)$ for any *value* v of these types. The values of type $⋵A$ 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 ⋵, 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, it divides the judgement into variables (to the left of $\checkmark_\theta$) received before the tick, and everything else, which happens after the tick. For example, 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. With this 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: 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. It is therefore not considered a value. Rather, $delay_\theta\ t$ 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 1), 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 been received. The type of $select\ v_1\ v_2$ reflects the three possible cases for such a tick: It could be in one of the two clocks $cl(v_1)$ and $cl(v_2)$, but not the other, or it could be in both. For example, if the input is in $cl(v_1)$, but not $cl(v_2)$, 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, 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 values that do not need to be evaluated. For example, evaluating $delay_{cl(t)}\ (adv\ t)$ would require evaluating t twice: first for evaluating the clock, and then to evaluate the term itself. Elimination of $\Diamond$ 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 proves 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 2. This transformation is also the basis for an implementation of *Async RaTT* as an embedded language in Haskell (Bahr et al., 2024).

2.2 Stable types and fixed points

General values in *Async RaTT* can contain references to time-dependent data, such as delayed computations stored in the heap. One of the main purposes of the type system

is to prevent such references to be dereferenced at times in the future when a delayed computation has been deleted from the heap. For this reason, arbitrary data should not be kept across time steps. This is reflected in the typing rule for variable introduction, which prevents general variables to be introduced across ticks.

For some types, however, values can not contain such references. We refer to these types as *stable types* and the grammar defining them is given in Figure 1. The two kinds of types that are explicitly not stable are delayed types $\textcircled{\exists} A$ and function types $A \to B$. Intuitively speaking, a value v of type $\textcircled{\exists} A$ 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 $\textcircled{\exists} A$.

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 hypothesis 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 $\vdash_\Delta box\, wait_\kappa : \Box(\textcircled{\exists} A)$ 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:

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

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 $fix\, x.t$ has type $\textcircled{\forall} A$, which means that the recursive definition can be unfolded to produce a term of type A any time in the future, but not now. This is ensured through the elimination rule for $\textcircled{\forall}$ which allows it to be advanced using a tick on any clock 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.

To understand the difference between the two later 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 $Fix\, \alpha.A$ are guarded recursive types that unfold to $A[\textcircled{\exists}(Fix\, \alpha.A)/\alpha]$ via the terms *into* and *out*. The most important of these types is $Sig\, A$ defined as $Fix\, \alpha.(A \times \alpha)$, which unfolds to $A \times \textcircled{\exists}(Sig\, A)$. 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 stable signal in *Async RaTT*:

$$box\left(fix\, x.delay_{cl(wait_\kappa)}\ (into\ \ (adv\, wait_\kappa, adv_\forall\, x))\right) : \square(\textcircled{\exists}(Sig\ A))$$

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

Besides all these constructions, *Async RaTT* also has a number of standard constructions from functional programming: sum types, product types, natural numbers and function types. The typing rules for these are completely standard.

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 θ when using $delay_\theta$ as it can be inferred from the context. This syntax can be easily elaborated into the *Async RaTT* calculus as described in section 3.6.

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 $s :: t : Sig\ A$.

We start with one of the simplest signal combinator:

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

The *map* combinator takes a stable function f and applies it pointwise to a given signal. The fact that f is of type $\square(A \to B)$ 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 $\square$ modality also has an intuitive justification: The function will be applied to values of the input signal arbitrarily far into the future, but a closure of type $A \to B$ may contain references to delayed computations that may have been garbage collected in the future.

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

$$scan : stable\ B \Rightarrow \square\ (B \to A \to B) \to B \to Sig\ A \to Sig\ B$$
$$scan\ f\ acc\ (a :: as) = acc' :: delay\ (scan\ f\ acc'\ (adv\ as))$$
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 $\Rightarrow$ notation to delineate such constraints from the type signature.

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

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

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

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

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

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

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

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

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

$$\begin{aligned}&\mathit{mapAwait} : \Box\ (A \to B) \to \Diamond\ (\mathit{Sig}\ A) \to \Diamond\ (\mathit{Sig}\ B)\\&\mathit{mapAwait}\ f\ d = \mathit{delay}\ (\mathit{map}\ f\ (\mathit{adv}\ d))\end{aligned}$$

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:

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

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

$$\begin{aligned}&\mathit{jump} : \Box\ (A \to (1 + \mathit{Sig}\ A)) \to \mathit{Sig}\ A \to \mathit{Sig}\ A\\&\mathit{jump}\ f\ (x :: \mathit{xs}) = \mathbf{case}\ \mathit{unbox}\ f\ x\ \mathbf{of}\\&\qquad\qquad\qquad\qquad \mathit{in}_1\ ()\ .\ x :: \mathit{delay}\ (\mathit{jump}\ f\ (\mathit{adv}\ \mathit{xs}))\\&\qquad\qquad\qquad\qquad \mathit{in}_2\ \mathit{xs}'.\ \mathit{xs}'\end{aligned}$$

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}. In the following section, we see more examples of combinators that combine several signals.

3.2 Concurrent Signal Combinators

The combinators we 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:

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

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 $in_1(in_1(s, t))$, i.e., the case that the left clock ticked first. The *interleave* combinator uses *select* in order to wait until at least one of the input signals ticks, and then updates the output signal accordingly. In case that both signals tick simultaneously, the provided merging function f is applied. For example, f could just always use the value of the first signal or take the average. Note that the produced signal combines the clocks of the input signals, i.e., it ticks whenever either of the input signals ticks.

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

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

$$\begin{array}{l} \quad \mathit{Right}\ \mathit{xs}'\ \mathit{ys}'.\ \mathit{zip}\ (x :: \mathit{xs}')\ \mathit{ys}' \\ \quad \mathit{Both}\ \ \mathit{xs}'\ \mathit{ys}'.\ \mathit{zip}\ \mathit{xs}'\ \mathit{ys}') \end{array}$$

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:

$$\begin{array}{l} \mathit{switch} : \mathit{Sig}\ A \to \text{∃} (\mathit{Sig}\ A) \to \mathit{Sig}\ A \\ \mathit{switch}\ (x :: \mathit{xs})\ d = x :: \mathit{delay}\ (\textbf{case}\ \mathit{select}\ \mathit{xs}\ d\ \textbf{of} \\ \qquad\qquad\qquad\qquad \mathit{Left}\ \ \mathit{xs}'\ d'.\ \mathit{switch}\ \mathit{xs}'\ d' \\ \qquad\qquad\qquad\qquad \mathit{Right}\ \_\ \ d'.\ d' \\ \qquad\qquad\qquad\qquad \mathit{Both}\ \ \mathit{xs}'\ d'.\ d') \end{array}$$

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:

$$\begin{array}{l} \mathit{switchS} : \mathit{stable}\ A \Rightarrow \mathit{Sig}\ A \to \text{∃} (A \to \mathit{Sig}\ A) \to \mathit{Sig}\ A \\ \mathit{switchS}\ (x :: \mathit{xs})\ d = x :: \mathit{delay}\ (\textbf{case}\ \mathit{select}\ \mathit{xs}\ d\ \textbf{of} \\ \qquad\qquad\qquad\qquad \mathit{Left}\ \ \ \ \mathit{xs}'\ \ d'.\ \mathit{switchS}\ \mathit{xs}'\ d' \\ \qquad\qquad\qquad\qquad \mathit{Right}\ \ \ \ \_\ \ \ d'.\ d'\ x \\ \qquad\qquad\qquad\qquad \mathit{Both}\ (x' :: \mathit{xs}')\ d'.\ d'\ x') \end{array}$$

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

$$\begin{array}{l} \mathit{switchR} : \mathit{stable}\ A \Rightarrow \mathit{Sig}\ A \to \text{∃} (\mathit{Sig}\ (A \to \mathit{Sig}\ A)) \to \mathit{Sig}\ A \\ \mathit{switchR}\ \mathit{sig}\ \mathit{steps} = \mathit{switchS}\ \mathit{sig} \\ \quad (\mathit{delay}\ (\textbf{let}\ \mathit{step} :: \mathit{steps}' = \mathit{adv}\ \mathit{steps}\ \textbf{in} \lambda\, x.\ \mathit{switchR}\ (\mathit{step}\ x)\ \mathit{steps}')) \end{array}$$

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

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

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

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

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

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 text field changes to $\{toggle\}$ as it now ignores the 'up' button. We will illustrate the run-time 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 – 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 *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:

$$
\begin{aligned}
&resetSig : \Circled{\exists}\ (Sig\ (Nat \times Nat \to Nat \times Nat))\\
&resetSig = mapAwait\ (box\ (\lambda\ ()\ (\_, max).\ (0, max)))\ sigAwait_{reset}\\
&setMaxSig : \Circled{\exists}\ (Sig\ (Nat \times Nat \to Nat \times Nat))\\
&setMaxSig = mapAwait\ (box\ (\lambda\ max'\ (n, \_).\ (min\ n\ max', max')))\ sigAwait_{max}
\end{aligned}
$$

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 using function composition, i.e., if both signals produce a function at the same time we sequentially compose them:

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

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:

$$
\begin{aligned}
&restartSig :: \Circled{\exists}\ (Sig\ (Nat \times Nat \to Sig\ (Nat \times Nat)))\\
&restartSig = mapAwait\ (box\ (\lambda f.\ start\ \circ f))\ inputSig\\
&timerMaxSig :: Sig\ (Nat \times Nat)\\
&timerMaxSig = switchR\ (start\ (0, 100))\ restartSig\\
&timerSig : Sig\ Nat\\
&timerSig = map\ (box\ (\lambda\ (n, \_).\ n))\ timerMaxSig
\end{aligned}
$$

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

```
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. 3. Integral and derivative signal combinators.

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 4.1):

```
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 3 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 won't change until the underlying signal has a non-zero value. Hence, we don't have to sample every time the *sample* channel ticks.

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

3.6 Elaboration of surface syntax into core calculus

To illustrate how the surface language used in this section 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* as it 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 these top-level definitions to be implicitly boxed when defined and unboxed when used later on. That is, these definitions are translated as follows to the core calculus:

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

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 prove the latter, the evaluation semantics uses a store in which both external inputs and delayed computations are stored. Delayed computations are garbage collected as soon as the data on which they depend has arrived. In this fashion, *Async RaTT* avoids implicit space leaks by construction, provided we can prove that the operational semantics never gets stuck.

4.1 Evaluation semantics

Figure 4 defines the evaluation semantics as a deterministic big-step operational semantics. We write $\langle t; \sigma \rangle \Downarrow^{\iota} \langle v; \tau \rangle$ to denote that when given a term t, a *store* σ, and an *input buffer* ι, the machine computes a value v and a new store τ. During the computation, the machine may defer computations into the future by storing unevaluated terms in the store σ to be retrieved and evaluated later. Conversely, the machine may also retrieve terms whose evaluation have been deferred at an earlier time and evaluate them now. In addition, the machine may read the new value of the most recently updated push channel from the store σ and read the current value of any buffered channel from the input buffer ι.

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 actual clocks (i.e., sets of push channels). When discussing the operational semantics, we have to be more precise about the distinction between these two. Therefore, we use θ to refer to clock expressions and Θ to refer to clocks. When the machine encounters a clock expression θ, it has to evaluate θ to a corresponding clock $|\theta|$. This occurs, for example, in the semantics for $delay_\theta$, which we return to 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 $cl(l) = \Theta$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. Operational semantics.

A clock Θ is a finite set of push channels drawn from Δ, and 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.

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 $\eta_N \in Heap^{\kappa}$. 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 indicates that the input channel κ has been updated to the new value v. The machine can thus safely resume computations from η_N since the data that the delayed computations in η_N were waiting for has arrived.

Let's first consider the semantics for *delay*: To allocate fresh locations in the store, we assume a function *alloc*, which, if given a clock Θ and a store η_L or $\eta_N\ \langle \kappa \mapsto v \rangle\ \eta_L$, produces a location $l \notin dom\,(\eta_L)$ with $cl\,(l) = \Theta$. The semantics of *delay* 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 $l \mapsto t$. The machine also has to evaluate the clock expression θ to a clock $|\theta|$ defined as follows:

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

Also *never* allocates a fresh heap location l, but it doesn't 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 semantics 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

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

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

$$\text{OUTPUT-END}\ \frac{}{\langle \cdot; \eta_N \langle \kappa \mapsto v\rangle\, \eta_L; \iota\rangle \Longrightarrow \langle \cdot; \eta_L; \iota\rangle}$$

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

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

Fig. 5. Reactive semantics.

the $adv_{\forall}$ combinator for $\textcircled{\forall}$ 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 $fix\, x.t$.

The evaluation semantics of the remaining language constructs is entirely standard.

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

The operational semantics of a reactive program is described by the machine in Figure 5. 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 pair $\langle N; \sigma; \iota\rangle$, where N is a sequence of the form $x_1 \mapsto l_1, \ldots, x_n \mapsto l_n$ that maps

all output channels $x_i \in dom\,(\Gamma_{out})$ to heap locations. That is, N records for each output channel the location of the delayed computation that will produce the next value of the output channel as soon as it needs updating.

The machine can make three kinds of transitions:

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

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 initial transition, which initialises the values of all output channels, the machine alternates between input transitions, each of which updates the value of an input channel and possibly the input buffer (if the new input is on a buffered channel), and output transitions, each of which provides new values for all 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 that 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 N, whether it should be advanced because it depends on κ (OUTPUT-COMPUTE) or should remain untouched because it does not depend on κ (OUTPUT-SKIP). Only in the OUTPUT-COMPUTE case a new output value for x is produced. In the end, the output transition performs the desired garbage collection that deletes both the now heap η_N and the input value v (OUTPUT-END). This also means that the updates performed by OUTPUT-COMPUTE, are not only possible (because the required data arrived), but also necessary (because both the input data and the delayed computations they depend on will be gone after this output transition of the machine).

4.3 Example

To see the operational semantics in action, we reconsider 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:

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

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 $\mathit{sigAwait}'_{\kappa}$ and scan' for the *dfix* versions of $\mathit{sigAwait}_{\kappa}$ and *scan* from section 3.1.

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

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

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

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

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

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

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

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

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

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

two transitions:

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

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

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:

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

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 set 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

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

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

While a reactive program will always produce a set of output values O_{i+1} for each incoming input value $\kappa_i \mapsto v_i$, this set may be empty. This happens if none of the heap locations in N_i depends on the input κ_i, i.e., if $\kappa_i \notin cl(l)$ for all $x \mapsto l \in N_i$. As we will see in Proposition 4.3, this will necessarily be the case for inputs $\kappa :_b A \in \Delta$ that are buffered-only.

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

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

each set of output values O_n only depends on the initial input buffer ι_0 and previously received input values $\kappa_i \mapsto v_i$ with $i < n$. This property follows from Theorem 4.1 and the fact 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* $c_1 = c_2$.

Hence, O_n is uniquely determined by ι_0 and $\kappa_i \mapsto v_i$ for all $i < n$.

4.4.3 Implicit space leaks

The operational semantics of *Async RaTT* is formulated in such a way so that after each pair of input/output transitions

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

all heap locations l in η that depend on κ, i.e., those with $\kappa \in cl(l)$, are moved into η_N after the input transition, and then η_N is garbage collected after 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 $\langle N; \eta_N \langle \kappa \mapsto v \rangle \eta_L; \iota \rangle$ with $\kappa \in cl(N(x))$. However, the typing system allows us to give two useful *static* criteria for when $\kappa \notin cl(N(x))$ 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 4.3 (buffered signal independence)**.** *If $t : \Delta \Rightarrow \Gamma_{out}$ is a reactive program and*

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

is a well-typed transition sequence for t, then O_{i+1} is empty whenever $\kappa_i :_b A \in \Delta$ for some A.

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

Theorem 4.4 (push signal independence)**.** Suppose $(t_1, \dots, t_n) : \Delta \Rightarrow \Gamma_{out}$ 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 $\Delta' \subset \Delta$. Given any well-typed transition sequence for $(t_1, \dots, t_m)$

$$\langle (t_1, \dots, t_m) ; \iota_0 \rangle \overset{O_0}{\Longrightarrow} \langle N_0; \eta_0; \iota_0 \rangle \overset{\kappa_0 \mapsto v_0}{\Longrightarrow} \langle N_0; \sigma_0; \iota_1 \rangle \overset{O_1}{\Longrightarrow} \langle N_1; \eta_1; \iota_1 \rangle \overset{\kappa_1 \mapsto v_1}{\Longrightarrow} \dots$$

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 4.3, and Theorem 4.4. 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 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 reactive semantics is indeed sound.

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

Firstly, time may pass, which means that we have fewer time steps left, i.e., $n > n'$. Secondly, the machine performs garbage collection on the store σ. The following 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 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 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 may also introduce an input value $\kappa \mapsto v$.

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

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

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

Finally, before we can give the definition of the Kripke logical relation, we need to semantically capture the notion of input independence that is needed both for the operational semantics of *select* and the signal independence properties (Proposition 4.3 and Theorem 4.4). 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 6. In addition to the aspects discussed above, it is parameterised by the context Δ and distinguishes between the value relation $\mathcal{V}_\Delta[\![A]\!](w)$ and the term relation $\mathcal{T}_\Delta[\![A]\!](w)$. The two relations are defined by well-founded recursion by the lexicographic ordering on the tuple $(n, |A|, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$|\alpha| = |\exists A| = |\forall A| = |1| = |Nat| = 1$$
$$|A \times B| = |A + B| = |A \to B| = 1 + |A| + |B|$$
$$|\Box A| = |Fix\,\alpha.A| = 1 + |A|$$

Note that in the definition for $\mathcal{V}_\Delta[\![\exists A]\!](n+1, \sigma)$, 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. That means we need a corresponding logical relation for contexts as well, which is given at the bottom of Figure 6. The interpretation of $\checkmark_\theta$ in a context is quite technical, but is essentially determined by the interpretation of $\exists$ 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, since any closed term $\vdash_\Delta t : A$ can 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:

$$\frac{\Gamma^\bigcirc, \checkmark_\theta \vdash_\Delta t : A}{\Gamma \vdash_\Delta delay_\theta\, t : \exists A} \tag{5.1}$$

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 $\exists A$. To capture this pattern, we define the notion of a *later type*, which is any type of the form $\exists A$ 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

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

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

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

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. 6. Logical relation.

a later type. For example, if

$$\Gamma = x_1 : \mathit{1} \to Nat, x_2 : \circledE Nat, x_3 : \circledV(Nat \to Nat), \checkmark_{cl(x_2)}, x_4 : \circledE Nat, x_5 : Nat, x_6 : \mathit{1} \to Nat$$

then $\Gamma^{\bigcirc} = x_3 : \circledV(Nat \to Nat), x_4 : \circledE Nat, x_5 : Nat$.

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 transform any typing derivation so that it uses the revised typing rule (5.1) 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 a 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 case for *box* and *fix* follows immediately from the fact that $(\Gamma, \Gamma')^{\Box} = (\Gamma^{\bigcirc}, \Gamma')^{\Box}$.
- The remaining cases follow immediately from the induction hypothesis. The case for *delay* additionally requires the property that $\Gamma, \Gamma' \vdash_\Delta \theta : Clock$ implies $\Gamma^{\bigcirc}, \Gamma' \vdash_\Delta \theta : Clock$, 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 the preservation under the Kripke preorder $\lesssim$:

Lemma 5.3. *Let $n \geq n'$, and $\sigma \sqsubseteq^{\Delta}_{\checkmark} \sigma'$.*

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

Proof (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). ■

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 $t \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma)$, and $\sigma' \sqsupseteq^{\Delta}_{\checkmark} \sigma$ 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 on 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). ∎

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

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

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

Proof 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 $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n, \sigma')$. In addition, Lemma 5.2 allows us to assume the revised typing rule (5.1) for *delay*.

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. ∎

Note that the proof uses Lemma 5.2 to transform the typing derivation on the fly so that it effectively uses rule (5.1) for *delay*.

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 4.3, and Theorem 4.4.

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:

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

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 notion of well-typedness for input buffers, input values, and output values, which are presented in 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 $\vdash_\Delta v : A$.*

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 $\langle t; \iota \rangle \overset{O}{\Longrightarrow} \langle N; \eta; \iota \rangle$ such that $\langle N; \eta; \iota \rangle \in S_n$ and $\vdash O : \Gamma_{out}$.
(iii) If $\langle N; \eta; \iota \rangle \in S_{n+1}$ and $\vdash \kappa \mapsto v : \Delta$, then there is a sequence of two transitions $\langle N; \eta; \iota \rangle \overset{\kappa \mapsto v}{\Longrightarrow} \langle N; \sigma; \iota' \rangle \overset{O}{\Longrightarrow} \langle N'; \eta'; \iota' \rangle$ such that $\langle N'; \eta'; \iota' \rangle \in S_n$ and $\vdash O : \Gamma_{out}$.

Proof

(i) We need to show that $t \in \mathcal{T}_\Delta[\![Prod\ (\Gamma_{out})]\!]\ (n, \emptyset)$. Since $t : \Delta \Rightarrow \Gamma_{out}$, we know that $\vdash_\Delta t : Prod\ (\Gamma_{out})$. Hence, by Theorem 5.8, we have that $t \in \mathcal{T}_\Delta[\![Prod\ (\Gamma_{out})]\!]\ (n, \emptyset)$.
(ii) Let $\langle t; \iota \rangle \in I_n$. We thus have $t \in \mathcal{T}_\Delta[\![Prod\ (\Gamma_{out})]\!]\ (n, \emptyset)$. Hence, $\langle t; \emptyset \rangle \Downarrow^\iota \langle (v_1 :: w_1, \ldots, v_k :: w_k)\ ; \eta \rangle$ with $v_i \in \mathcal{V}_\Delta[\![A_i]\!]\ (n, \eta)$ and $w_i \in \mathcal{V}_\Delta[\![\circledexists (Sig\ A_i)]\!]\ (n, \eta)$. Since $Sig\ A_i$ is not a value type, w_i cannot be of the form $wait_\kappa$. 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 $\langle N; \eta; \iota\rangle \overset{\kappa\mapsto v}{\Longrightarrow} \langle N; \sigma; \iota'\rangle$, where $\sigma = [\eta]_{\kappa\in}\ \langle \kappa \mapsto v\rangle\ [\eta]_{\kappa\notin}$, $\iota' = \iota[\kappa \mapsto v]$ if $\kappa \in dom\ (\iota)$, and $\iota' = \iota$ otherwise. To construct the second transition step, we show the following more general property:

$$\begin{array}{l}\text{If } \sigma' \sqsupseteq \sigma, N_1 \subseteq N, \text{ then } \langle N_1; \sigma'; \iota'\rangle \overset{O}{\Longrightarrow} \langle N_2; \eta'; \iota'\rangle\ , \\ \text{such that } \langle N_2; \eta'; \iota'\rangle \in S_n, gc(\sigma') \sqsubseteq \eta', \text{ and } \vdash O : \Gamma_{out}.\end{array} \tag{5.2}$$

From (5.2), we then obtain that $\langle N; \sigma; \iota'\rangle \overset{O}{\Longrightarrow} \langle N'; \eta'; \iota'\rangle$, with $\langle N'; \eta'; \iota'\rangle \in S_n$ and $\vdash O : \Gamma_{out}$. We conclude this proof by proving (5.2) by induction on the size of N_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 $N_1 = \cdot$. Then, by definition, we get the transition $\langle \cdot; \sigma'; \iota'\rangle \Longrightarrow \langle \cdot; gc(\sigma'); \iota'\rangle$, 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 $N_1 = x \mapsto l, N_1'$ with $\kappa \notin cl(l)$ and $x : A \in \Gamma_{out}$. By induction hypothesis, we obtain a transition $\langle N_1'; \sigma'; \iota'\rangle \overset{O}{\Longrightarrow} \langle N_2'; \eta'; \iota'\rangle$, with $\langle N_2'; \eta'; \iota'\rangle \in S_n$, $gc(\sigma') \sqsubseteq \eta'$, and $\vdash O : \Gamma_{out}$. By definition, we obtain the transition $\langle x \mapsto l, N_1'; \sigma'; \iota'\rangle \overset{O}{\Longrightarrow} \langle x \mapsto l, N_2'; \eta'; \iota'\rangle$. Since $\langle N; \eta; \iota\rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n+1, \eta)$. Because of $\kappa \notin cl(l)$, we can conclude that $[\eta]_{cl(l)} = \left[[\eta]_{\kappa\notin}\right]_{cl(l)}$, which, by Lemma 5.7, means that $l \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n+1, [\eta]_{\kappa\notin})$. Since $[\eta]_{\kappa\notin} \sqsubseteq \eta_L = gc(\sigma') \sqsubseteq \eta'$, we have by Lemma 5.3 that $l \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n, \eta')$ and thus $\langle x \mapsto l, N_2'; \eta'; \iota'\rangle \in S_n$.
- Let $N_1 = x \mapsto l, N_1'$ with $\kappa \in cl(l)$ and $x : A \in \Gamma_{out}$. Since $\langle N; \eta; \iota\rangle \in S_{n+1}$, we know that $l \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n+1, \eta)$, 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 $l' \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n, \sigma'')$. By induction hypothesis, we obtain a transition $\langle N_1'; \sigma''; \iota'\rangle \overset{O}{\Longrightarrow} \langle N_2'; \eta'; \iota'\rangle$, with $\langle N_2'; \eta'; \iota'\rangle \in S_n$, $gc(\sigma'') \sqsubseteq \eta'$, 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 $\langle x \mapsto l, N_1'; \sigma'; \iota'\rangle \overset{x\mapsto v', O}{\Longrightarrow} \langle x \mapsto l', N_2'; \eta'; \iota'\rangle$. By Lemma 5.3 and Lemma 5.6, we have that $l' \in \mathcal{V}_\Delta[\![\ominus(Sig\ A)]\!]\ (n, \eta')$, and thus we also have $\langle x \mapsto l', N_2'; \eta'; \iota'\rangle \in S_n$. In addition, since $v' \in \mathcal{V}_\Delta[\![A]\!]\ (n, \sigma'')$ implies $\vdash_\Delta v' : A$ 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:

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

with $\vdash O_i : \Gamma_{out}$ 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 4.3 (buffered signal independence)] By Lemma 5.10 any input transition $\langle N_i; \eta_i; \iota_i \rangle \overset{\kappa_i \mapsto v_i}{\Longrightarrow} \langle N_i; \sigma_i; \iota_{i+1} \rangle$ in a sequence starting from $\langle t; \iota_0 \rangle$ will be in S_1. In particular, this means that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists}(Sig\ A)]\!]\ (1, \eta_i)$ for any $x \mapsto l \in N_i$ with $x : A \in \Gamma_{out}$, which in turn means that $cl(l) \in Loc_\Delta$. Hence, $\kappa \notin cl(l)$, and so O_{i+1} is empty. ∎

For the proof of Theorem 4.4, we first define the following sets of machine states in the context of a partial map D that maps variables x to input contexts D_x:

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

Machine states in T_n^D are maintained during the execution of the machine:

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

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

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

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

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 $\eta(l) = [\eta]_{\kappa \in}\ (l) = \sigma'(l)$, by Lemma 4.2, $\sigma''' = \sigma''$ and $l'' = l'$. From this we conclude that

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

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 4.4 (push signal independence)] The initialisation transition

$$\langle (t_1, \ldots, t_m)\ ; \iota_0 \rangle \overset{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m}{\Longrightarrow} \langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m; \eta_0; \iota_0 \rangle = \langle N_0; \eta_0; \iota_0 \rangle$$

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, $l_k \in$

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

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

Fig. 7. Revised typing rules for *Full Async RaTT*.

$\mathcal{V}_\Delta[\![\circledcirc(Sig\ B_k)]\!]\ (n, \eta_0)$ for all n, and $k = 1, \ldots, m$. So $\langle x_1 \mapsto l_1, \ldots, x_m \mapsto l_m; \eta_0; \iota_0\rangle \in T_n^D$ for all n, where $D_{x_j} = \Delta'$ and $D_{x_k} = \Delta$, for $k \neq j$. By n applications of Lemma 5.11, we thus obtain that $\langle N_n; \eta_n; \iota_n\rangle \in T_1^{\Delta''}$. In particular, $N_n(x_j) = l'$ for some $l' \in \mathcal{V}_{\Delta'}[\![\circledcirc(Sig\ A_j)]\!]\ (1, \eta_n)$. 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 $\kappa_i \in dom\ (\Delta')$. ■

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 arbitrarily 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, these restrictions *are* 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 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., 2024).

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 2. The new typing rules that replace these in *Full Async RaTT* are shown in Figure 7. To distinguish the type system from that of Figure 2, 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 8. 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 $s(\!|t|\!)$.

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

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

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

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

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

Fig. 8. Rewrite rules for the transformation of *Full Async RaTT* terms to *Async RaTT* terms.

All rewrite rules replace occurrences of subterms of the form $delay_\theta\ s$, where θ is not a value, i.e., 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 occurrences of *adv* or *select* that consumes the tick $\checkmark_\theta$. Note that the rules are not mutually exclusive, 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

$$\lambda x.delay_{cl(x)}\ (delay_{cl(adv\ x)}\ (suc\ (adv\ (adv\ x)))) \tag{6.1}$$

of type $\Im(\Im Nat) \to \Im(\Im Nat)$. 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 $adv\ x$:

$$\lambda x.delay_{cl(x)}\ (let\ y = adv\ x\ in\ delay_{cl(y)}\ (suc\ (adv\ y))) \tag{6.2}$$

Rule (T1) captures exactly this transformation. To this end, the rewrite rule use 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 (6.1), we can see that we obtain the term (6.2), by performing the following subsitution:

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

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 occurrence of *adv* or *select* that consumes the tick on θ. For example, consider the following *Full Async RaTT* term of type $(Nat \to \Circledcirc 1) \to \Circledcirc\,(\Circledcirc Nat \to \Circledcirc Nat)$:

$$\lambda x.delay_{cl(x\,0)\sqcup cl(x\,1)\sqcup cl(x\,2)}\ (\lambda y.delay_{cl(y)}\ (suc\ (adv\ y))) \tag{6.3}$$

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. Since 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 (6.3) into a term that first evaluates $x\,0$, $x\,1$, and $x\,2$ to values before constructing the clock:

$$\begin{aligned}&\lambda x.let\ z_0 = x\,0\ in\ let\ z_1 = x\,1\ in\ let\ z_2 = x\,2\ in\\ &\quad delay_{cl(z_0)\sqcup cl(z_1)\sqcup cl(z_2)}\ (\lambda y.delay_{cl(y)}\ (suc\ (adv\ y)))\end{aligned}$$

To understand the final rewrite rule, we need to introduce some terminology: We use the notation T for a clock expression with a single hole $[]$ and 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 $[]$, which must not be the scope of *delay*, *adv*, *select*, *box*, or *fix*. More precisely, K is generated by the following grammar:

$$\begin{aligned}K ::= {} & [] \mid K\ t \mid t\ K \mid \lambda x.K \mid let\ x = K\ in\ t \mid let\ x = t\ in\ K \mid (K, t) \mid (t, K) \mid in_1\ K \mid in_2\ K \mid \pi_1\ K\\ & \mid \pi_2\ K \mid case\ K\ of\ in_1\ x.s; in_2\ x.t \mid case\ s\ of\ in_1\ x.K; in_2\ x.t \mid case\ s\ of\ in_1\ x.t; in_2\ x.K\\ & \mid suc\ K \mid into\ K \mid out\ K \mid unbox\ K \mid rec_{Nat}(K, x\ y.t, u) \mid rec_{Nat}(s, x\ y.K, u)\\ & \mid rec_{Nat}(s, x\ y.t, K)\end{aligned}$$

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 $K[adv\ s]$ nor $K[select\ s\ s']$. 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 K. Namely, $t(\!|s|\!)$ is the result of repeatedly applying to t the rewrite relation defined by $K[adv\ s'] \longrightarrow K[s]$ and $K[select\ s_1\ s_2] \longrightarrow K[s]$.

Returning to the example, we can see that (T4) can be applied to (6.3) since the term $\lambda y.delay_{cl(y)}\ (suc\ (adv\ y))$ is tick-closed: The occurrence of $adv\ y$ is guarded by $delay_{cl(y)}$ and thus $\lambda y.delay_{cl(y)}\ (suc\ (adv\ y))$ is not of the form $K[adv\ y]$.

As mentioned above, sometimes more than one rewrite rule is applicable. For example, if we have a term of the form $delay_{cl(t)}\ s$ where s is tick-closed and t is not a value, then both (T1) and (T4) apply. However, the outcome of applying either of these rules is the same, because $s(\!|t|\!) = s$ whenever s is tick-closed.

6.2 Soundness of the program transformation

We show that the rewrite relation $\longrightarrow$ is strongly normalising 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 an *Async RaTT* term of the same type:

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

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

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

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

6.2.1 Exhaustiveness

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

Lemma 6.2. *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. ∎

We say that a clock expression θ is a value if any term t occurring in θ is a value. Any well-formed value clock in *Full Async RaTT* is also a well-formed clock in *Async RaTT*:

Lemma 6.3. *If* $\Gamma \Vdash_\Delta \theta : Clock$ *and* θ *is a value, then* $\Gamma \vdash_\Delta \theta : Clock$.

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

Moreover, exhaustively applying $\longrightarrow$ ensures that all clocks are indeed values:

Lemma 6.4. *If* $\Gamma, \checkmark_\theta \Vdash_\Delta t : A$ *and* $delay_\theta\ t \not\longrightarrow$, *then* θ *is a value.*

Proof We consider two cases:

- t is tick-closed: Then $delay_\theta\ t \not\longrightarrow$ implies that θ is a value.
- t is not tick-closed: Hence, t is of the form $K[adv\ s]$ or $K[select\ t_1\ t_2]$, which by Lemma 6.2 means that $\theta = cl(s)$ or $\theta = cl(t_1) \sqcup cl(t_2)$, respectively. Hence, $delay_\theta\ t \not\longrightarrow$ implies that θ is a value. ∎

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 6.5 (exhaustiveness)**.** *If* $\Gamma \vdash_\Delta$, $\Gamma \Vdash_\Delta t : A$, *and* $t \not\rightarrow\!\!\!\!\rightarrow$, *then* $\Gamma \vdash_\Delta t : A$.

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

- $\Gamma \Vdash_\Delta \mathit{delay}_\theta\, t : \textcircled{\exists} A$: Hence, $\Gamma, \checkmark_\theta \Vdash_\Delta t : A$ and $\Gamma \Vdash_\Delta \theta : \mathit{Clock}$. By Lemma 6.3 and Lemma 6.4, 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 $\Gamma \vdash_\Delta \mathit{delay}_\theta\, t : \textcircled{\exists} A$.
- $\Gamma, \checkmark_{cl(t)}, \Gamma' \Vdash_\Delta \mathit{adv}\, t : A$: Hence, $\Gamma \Vdash_\Delta t : \textcircled{\exists} A$. From the assumption $\Gamma, \checkmark_{cl(t)}, \Gamma' \vdash_\Delta$ we obtain that t is a value and that $\Gamma \vdash_\Delta$. Hence, we may apply the induction hypothesis to obtain that $\Gamma \vdash_\Delta t : \textcircled{\exists} A$. Since t is a value, we thus have $\Gamma, \checkmark_{cl(t)}, \Gamma' \vdash_\Delta \mathit{adv}\, t : A$.
- $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta \mathit{select}\, t_1\, t_2 : ((A_1 \times \textcircled{\exists} A_2) + (\textcircled{\exists} A_1 \times A_2)) + (A_1 \times A_2)$: 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*. ∎

6.2.2 *Subject reduction*

The proof of subject reduction makes use of the side condition that the rewrite rules in Figure 8 can only be applied to a term $\mathit{delay}_\theta\, s$ if $s \not\rightarrow\!\!\!\!\rightarrow$. As we have shown in section 6.2.1, this side condition ensures that any clocks in s must be values, which allows us to apply the following lemma:

Lemma 6.6. *If* $\Gamma, \checkmark_\theta, \Gamma' \Vdash_\Delta t : A$, $\Gamma \Vdash_\Delta \theta' : \mathit{Clock}$, *and* Γ' *contains a tick on a value clock, 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, $\Gamma' = \Gamma_1, \checkmark_{cl(s)}, \Gamma_2$ and $\Gamma, \checkmark_\theta, \Gamma_1 \Vdash_\Delta s : \textcircled{\exists} A$. To conclude $\Gamma, \checkmark_{\theta'}, \Gamma' \Vdash_\Delta t : A$, it remains to be shown that $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \textcircled{\exists} A$:
  - If s is a value, then it is either a variable or of the form wait_κ. In either case, we have that $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \textcircled{\exists} A$.
  - If s is not a value, then Γ_1 must still contain a tick on a value clock since $cl(s)$ is not a value. Hence, we can apply the induction hypothesis to obtain $\Gamma, \checkmark_{\theta'}, \Gamma_1 \Vdash_\Delta s : \textcircled{\exists} A$.
- 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 6.7. *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* $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|adv\, x|\!) : A$ *and* $\Gamma \Vdash_\Delta s : \Box\Diamond B$ *for some* B.
(ii) *If* $\theta = cl(t_1) \sqcup cl(t_2)$, *then* $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)\sqcup cl(t_2)}, \Gamma' \Vdash_\Delta t(\!|select\, x\, t_2|\!) : A$ *and* $\Gamma \Vdash_\Delta t_1 : \Box\Diamond B$ *for some* B.
(iii) *If* $\theta = cl(t_1) \sqcup cl(t_2)$, *then* $\Gamma, x : \Box\Diamond B, \checkmark_{cl(t_1)\sqcup cl(x)}, \Gamma' \Vdash_\Delta t(\!|select\, t_1\, x|\!) : A$ *and* $\Gamma \Vdash_\Delta t_2 : \Box\Diamond B$ *for some* B.
(iv) *If* $\theta = T[cl(s)]$ *and* t *tick-closed, then* $\Gamma, x : \Box\Diamond B, \checkmark_{T[cl(x)]}, \Gamma' \Vdash_\Delta t : A$ *and* $\Gamma \Vdash_\Delta s : \Box\Diamond B$ *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 $\Gamma \Vdash_\Delta s : \Box\Diamond B$ for some B. We proceed by induction on t:

- If t is of the form $box\, t'$, $delay_{\theta'}\, t'$, or $fix\, x.t'$, then $t(\!|adv\, x|\!) = t$. By weakening, we have $\Gamma, x : \Box\Diamond B, \checkmark_\theta, \Gamma' \Vdash_\Delta t(\!|adv\, x|\!) : A$. If t is of the form $box\, t'$ or $fix\, x.t'$, then $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|adv\, x|\!) : A$ follows immediately from the typing rules for *box* and *fix*. Otherwise, if $t = delay_{\theta'}\, t'$, then $\Gamma, x : \Box\Diamond B, \checkmark_\theta, \Gamma', \checkmark_{\theta'} \Vdash_\Delta t' : A'$ for some A' with $\Box\Diamond A' = A$. Then we can apply Lemma 6.4 to obtain that $\Gamma', \checkmark_{\theta'}$ contains a tick on a value clock. This allows us to apply Lemma 6.6 to obtain that $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)}, \Gamma', \checkmark_{\theta'} \Vdash_\Delta t' : A'$. Hence, $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|adv\, x|\!) : A$ follows.
- The case $t = select\, t_1\, t_2$ is impossible.
- If $t = adv\, s'$, then $t(\!|adv\, x|\!) = adv\, x$ and $A = B$. Hence, $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)}, \Gamma' \Vdash_\Delta t(\!|adv\, x|\!) : A$.
- The remaining cases all follow by induction hypothesis. ∎

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

Proposition 6.8 (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 $\Gamma \Vdash_\Delta t : A$ follows from Lemma 6.7. For example, in case of rule (T1), we have that $s = delay_{cl(t')}\, s''$, $t = let\, x = t'\, in\, delay_{cl(x)}(s''(\!|adv\, x|\!))$, and $s'' \not\longrightarrow$. Consequently, A is of the form $\Box\Diamond A'$ for some A' and $\Gamma, \checkmark_{cl(t')} \Vdash_\Delta s'' : A'$ and $\Gamma \Vdash_\Delta cl(t') : \mathit{Clock}$. By Lemma 6.7, we have that $\Gamma, x : \Box\Diamond B, \checkmark_{cl(x)} \Vdash_\Delta s''(\!|adv\, x|\!) : A'$ and $\Gamma \Vdash_\Delta t' : B$ for some B. Hence, the typing rules for *delay* and let bindings apply to yield $\Gamma \Vdash_\Delta t : A$. ∎

6.2.3 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-value clock expression $cl\,(t)$:

Proposition 6.9 (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 6.8, 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 6.2 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 to a corresponding *Async RaTT* program 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:

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

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

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* so that it evaluates its clock argument to a value:

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

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 that takes a signal of numbers and produces a new signal that simply repeats every other element of the input signal:

$$stutter : Sig\ Nat \to Sig\ Nat$$
$$stutter\ (x :: xs) = x :: delay\ (x :: delay\ (stutter\ (adv\ (tail\ (adv\ xs)))))$$

This definition elaborates into the following *Full Async RaTT* term:

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

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:

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

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

$$\langle x \mapsto l_2; l_1, l_2; \emptyset\rangle \overset{nat\mapsto 1}{\Longrightarrow} \langle x \mapsto l_2; l_1, l_2\ \langle nat \mapsto 1\rangle\ \emptyset; \emptyset\rangle \overset{x\mapsto 0}{\Longrightarrow} \langle x \mapsto l_3; l_3; \emptyset\rangle$$
$$\text{where} \quad l_3 \mapsto adv_\forall\ stutter'\ (adv\ (\pi_2\ (out\ (adv\ l_1))))$$

However, as we can already see, the delayed computation stored at l_3 references l_1, which has already been garbage collected. As a result, the machine gets stuck after the next input step:

$$\langle x \mapsto l_3; l_3; \emptyset\rangle \overset{nat\mapsto 2}{\Longrightarrow} \langle x \mapsto l_3; l_3\ \langle nat \mapsto 2\rangle\ \emptyset; \emptyset\rangle \Longrightarrow\!\!\!\!\!\!\not\quad$$

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 occurrence of *adv* in the term stored at l_3, which the evaluation semantics cannot support, due to its aggressive garbage collection.

But even if we don't allow for two nested occurrences of *delay*, we can still construct a (rather contrieved) *Full Async RaTT* program that cannot be directly run by the machine. This example is adaped from Bahr et al. (2019):

$$leaky : (1 \to 1) \to Sig\ A \to Sig\ A$$
$$leaky\ f\ (x :: xs) = x :: delay\ (leaky\ (\lambda\, y.\ adv\ (f\ ();xs);())\ (adv\ (f\ ();xs)))$$

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

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

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

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

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

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

After the next input transition, the machine gets stuck trying to evaluate the subterm $(\lambda y.adv\ (id\ ();l_1);())$ 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:

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

Perhaps, a solution to address this problem is that we simply have to hold 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 *Async RaTT* program 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's 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:

$$\begin{aligned} \langle x \mapsto l_4; l_3, l_4; \emptyset \rangle &\overset{nat \mapsto 2}{\Longrightarrow} \langle x \mapsto l_4; l_3, l_4 \, \langle nat \mapsto 1 \rangle \, \emptyset; \emptyset \rangle \overset{x \mapsto 2}{\Longrightarrow} \langle x \mapsto l_6; l_5, l_6; \emptyset \rangle \\ &\text{where} \quad l_5 \mapsto adv\ wait_{nat} :: adv_\forall\ sigAwait'_{nat} \\ &\qquad\quad l_6 \mapsto adv_\forall\ leaky'_T\ (\lambda y.adv\ l_5; ())\ (adv\ l_5) \end{aligned}$$

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 like the ones proved here for lack of space leaks. The first such operational guarantees were given by Krishnaswami et al. (2012) who describe a modal FRP language using linear types and allocation resources to statically bound

the memory used by a reactive program. The simpler, but less precise, idea of using an aggressive garbage collection technique for avoiding space leaks is due to Krishnaswami (2013). Krishnaswami's calculus used a dual context approach to programming with modal types. Bahr et al. (2019) recast these results in a Fitch-style modal calculus, the first in the RaTT family. This was later implemented in Haskell with some minor modifications (Bahr, 2022).

All the above calculi are based on a global notion of time, which in almost all cases is discrete. In particular, the modal operator $\bigcirc$ for time-steps in these calculi refers to the next time step on the global clock. One can of course also understand the step semantics of *Async RaTT* as operating on a global clock, but in our model each step is associated with an input coming from an input channel, and this allows us to define the delay modality $\textcircled{\exists}$ as a delay on a set of input channels. From the model perspective, $\textcircled{\exists}A$ 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, has it limitations, in particular 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 synchronisation primitive similar to *select* in *Async RaTT*. However, the programming primitives in λ_{widget} are very different from the ones use here. For example, λ_{widget} allows an element of $\Diamond A$ to be decomposed into a time and an element of A at that time, and much programming with $\Diamond$ uses this decomposition. There is also no delay type constructor $\bigcirc$, so $\textcircled{\exists}$ is not expressible: Unlike $\textcircled{\exists}A$, an element of $\Diamond A$ could give a value of type A already now. Graulund et al. provide a denotational semantics for λ_{widget}, but no operational semantics, and no operational guarantees as proved here.

Another approach to avoiding space leaks and non-causal reactive programs is to devise a carefully designed interface to manipulate signals such as Yampa (Nilsson et al., 2002) or FRPNow! (Ploeg and Claessen, 2015). Rhine (Bärenz and Perez, 2018) is a recent refinement of Yampa that annotates signal functions with type-level clocks, which allows the construction of complex dataflow graphs that combine subsystems running at different clock speeds. The typing discipline fixes the clock of each subsystem *statically* at compile time, since the aim of Rhine is 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 $Sig\ (Time \to A)$, i.e., at each tick of the clock we get a new function $Time \to A$ that describes the time-varying value of the signal until the next tick of the clock.

Futures, first implemented in MuliLisp (Halstead, 1985) and now commonly found in many programming languages under different names (promise, async/await, delay, etc.), provide a powerful abstraction to facilitate communication between concurrent computations. A value of type $Future\ A$ 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 *Future 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 $\Diamond$, either directly or by defining *Future* as a guarded recursive type $\mathit{Future}\ A \cong A + \Diamond(\mathit{Future}\ A)$ 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 *Future 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) used a linear typing discipline to obtain static memory bounds. In addition to such memory bounds, synchronous dataflow languages such as Esterel (Berry and Cosserat, 1985), Lustre (Caspi et al., 1987), and Lucid Synchrone (Pouzet, 2006) even provide bounds on runtime. Despite these strong guarantees, Lucid Synchrone affords a high-level, modular programming style with support for higher-order functions. However, to achieve such static guarantees, synchronous dataflow languages must necessarily enforce strict limits on the dynamic behaviour, disallowing both time-varying values of arbitrary types (e.g., we cannot have a stream of streams) and dynamic switching (i.e., no functionality equivalent to the *switch* combinator). Both Lustre and Lucid Synchrone have a notion of a clock, which is simply a stream of Booleans that indicates at each tick of the global clock, whether the local clock ticks as well.

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 $\Diamond$ for asynchronous delay can be used to annotate the runtime system with dependencies from output channels to input channels, ensuring that outputs are only recomputed when necessary. The examples of the integral and the derivative even show how the programmer can actively influence the update rate of output channels.

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

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

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

Appel, A. W. & McAllester, D. (2001) An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Trans. Program. Lang. Syst.* **23**(5), 657–683. 00283.

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

Bahr, P., Graulund, C. U. & Møgelberg, R. E. (2019) Simply RaTT: a Fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages*. **3**(ICFP), 1–27.

Bahr, P., Graulund, C. U. & Møgelberg, R. E. (2021) Diamonds are not forever: Liveness in reactive programming with guarded recursion. *Proceedings of the ACM on Programming Languages*. **5**(POPL), 1–28.

Bahr, P., Houlborg, E. & Rørdam, G. T. S. (2024) Asynchronous Reactive Programming with Modal Types in Haskell. Practical Aspects of Declarative Languages. Springer Nature Switzerland. pp. 18–36.

Bahr, P. & Møgelberg, R. E. (2023) Asynchronous Modal FRP. *Proceedings of the ACM on Programming Languages*. **7**(ICFP), 205:476–205:510.

Berry, G. & Cosserat, L. (1985) The ESTEREL synchronous programming language and its mathematical semantics. Seminar on Concurrency. Berlin, Heidelberg, DE. Springer Berlin Heidelberg. pp. 389–448.

Birkedal, L., Clouston, R., Mannaa, B., Møgelberg, R. E., Pitts, A. M. & Spitters, B. (2020) Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*. **30**(2), 118–138.

Bärenz, M. & Perez, I. (2018) Rhine: FRP with type-level clocks. Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell. New York, NY, USA. Association for Computing Machinery. pp. 145–157.

Caspi, P., Pilaud, D., Halbwachs, N. & Plaice, J. A. (1987) Lustre: A declarative language for real-time programming. Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York, NY, USA. ACM. pp. 178–188.

Cave, A., Ferreira, F., Panangaden, P. & Pientka, B. (2014) Fair Reactive Programming. Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego, California, USA. ACM. pp. 361–372.

Clouston, R. (2018) Fitch-style modal lambda calculi. Foundations of Software Science and Computation Structures. Cham. Springer. Springer International Publishing. pp. 258–275.

Davies, R. & Pfenning, F. (2001) A modal analysis of staged computation. *Journal of the ACM (JACM)*. **48**(3), 555–604.

Elliott, C. & Hudak, P. (1997) Functional reactive animation. Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA. ACM. pp. 263–273.

Elliott, C. M. (2009) Push-pull Functional Reactive Programming. Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. New York, NY, USA. ACM. pp. 25–36.

Graulund, C. U., Szamozvancev, D. & Krishnaswami, N. (2021) Adjoint reactive GUI programming.

FoSSaCS. pp. 289–309.
Halstead, R. H. (1985) Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*. **7**(4), 501–538.
Jeffrey, A. (2012) LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012. Philadelphia, PA, USA. ACM. pp. 49–60.
Jeffrey, A. (2014) Functional reactive types. 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). New York, NY, USA. ACM. pp. 54:1–54:9.
Jeltsch, W. (2012) Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*. **286**, 229–242.
Kiss, E. (2014) 7GUIs: A GUI programming benchmark. https://eugenkiss.github.io/7guis/tasks.
Krishnaswami, N. R. (2013) Higher-order Functional Reactive Programming Without Spacetime Leaks. Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. Boston, Massachusetts, USA. ACM. pp. 221–232.
Krishnaswami, N. R. & Benton, N. (2011) Ultrametric semantics of reactive programs. 2011 IEEE 26th Annual Symposium on Logic in Computer Science. Washington, DC, USA. IEEE Computer Society. pp. 257–266.
Krishnaswami, N. R., Benton, N. & Hoffmann, J. (2012) Higher-order functional reactive programming in bounded space. Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. Philadelphia, PA, USA. ACM. pp. 45–58.
Nakano, H. (2000) A modality for recursion. Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332). Washington, DC, USA. IEEE Computer Society. pp. 255–266.
Nilsson, H., Courtney, A. & Peterson, J. (2002) Functional reactive programming, continued. Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. New York, NY, USA. ACM. pp. 51–64.
Ploeg, A. v. d. & Claessen, K. (2015) Practical principled FRP: forget the past, change the future, FRPNow! Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. Vancouver, BC, Canada. Association for Computing Machinery. pp. 302–314. 00019.
Pnueli, A. (1977) The temporal logic of programs. Proceedings of the 18th Annual Symposium on Foundations of Computer Science. USA. IEEE Computer Society. pp. 46–57.
Pouzet, M. (2006) Lucid Synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*. **1**, 25.
Reppy, J. H. (1999) *Concurrent programming in ML*. Cambridge University Press. Cambridge, England.

A Proof of the Fundamental Property

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

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

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

Proof By a straightforward case analysis. ∎

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

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

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

Proof By a straightforward induction on the length of Γ using Lemma 5.3 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* $\gamma \in \mathcal{C}_{\Delta}[\![\Gamma]\!]\ (w)$, *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 $|\theta\gamma| \subseteq dom\ (\Delta)$. Hence, $(\theta \sqcup \theta')\gamma = \theta\gamma \sqcup \theta'\gamma$ is closed and $|(\theta \sqcup \theta')\gamma| = |\theta\gamma| \cup |\theta'\gamma| \subseteq dom\ (\Delta)$.
- $$\frac{\Gamma \vdash_{\Delta} v : \circledexists A}{\Gamma \vdash_{\Delta} cl\,(v) : Clock}$$

$\Gamma \vdash_\Delta v : \circledcirc A$ implies that $v\gamma \in \mathcal{V}_\Delta[\![\circledcirc A]\!]\,(w)$ (because either v is a variable or $v = wait_\kappa$ for some clock κ). Hence, $v\gamma \in Loc_\Delta \cup \{wait_\kappa \mid \kappa \in dom_p\,(\Delta)\}$ and thus $cl\,(v)\,\gamma$ is a closed clock expression and $|cl\,(v)\,\gamma| \subseteq dom_p\,(\Delta)$.

■

Lemma A.7. *Let $\eta_N \in Heap^\kappa$, $v \in \mathcal{V}_\Delta[\![\circledcirc A]\!]\,(n+1, (\eta_N, [\eta_L]_{\kappa\notin}))$, $\vdash \iota : \Delta$, $\vdash w : \Delta(\kappa)$ 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)**.** *Given $\Gamma \vdash_\Delta$, $\Gamma \vdash_\Delta t : A$, and $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma)$, then $t\gamma \in \mathcal{T}_\Delta[\![A]\!]\,(n, \sigma)$.*

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 s.t. $\langle t\gamma;\sigma\rangle \Downarrow^{\iota} \langle v;\sigma''\rangle$ and $v \in \mathcal{V}_\Delta[\![A]\!]\,(n,\sigma'')$. By Lemma 5.3 we may assume that $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma')$.

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

  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 $x \in dom_p\left(\gamma|_{\Gamma^{\square}}\right)$. By Lemma A.4, $\gamma|_{\Gamma^{\square}} \in \mathcal{C}_\Delta[\![\Gamma^{\square}]\!]\,(n,\emptyset)$. 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 $x\gamma \in \mathcal{V}_\Delta[\![A]\!]\,(n,\sigma)$.

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

  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, $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n,\sigma'')$ and thus

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

  Hence, we may apply the induction hypothesis to obtain $t\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!]\,(n,\sigma'')$. Since all elements in the range of γ are closed terms, $t\gamma[x \mapsto v] = (t\gamma)[v/x]$ and thus $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(n,\sigma'')$. Consequently, $\langle (t\gamma)[v/x];\sigma''\rangle \Downarrow^{\iota} \langle w;\sigma'''\rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!]\,(n,\sigma''')$. By definition, we thus have $\langle (\mathit{let}\, x = s\, \mathit{in}\, t)\gamma;\sigma'\rangle \Downarrow^{\iota} \langle w;\sigma'''\rangle$ 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 $\sigma' \sqsupseteq^{\Delta}_{\surd} \sigma$, $n' \leq n$, and $v \in \mathcal{V}_\Delta[\![A]\!]\,(n',\sigma')$, with the goal of showing $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(n',\sigma')$. By Lemma 5.3, $\gamma \in \mathcal{C}_\Delta[\![\Gamma]\!]\,(n',\sigma')$. Thus, by definition, $\gamma[x \mapsto v] \in \mathcal{C}_\Delta[\![\Gamma, x : A]\!]\,(n',\sigma')$. By induction, we then have that $t\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!]\,(n',\sigma')$. Since all elements in the range of γ are closed terms, $t\gamma[x \mapsto v] = (t\gamma)[v/x]$ and thus $(t\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(n',\sigma')$.

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

  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, $\sigma'' \sqsupseteq^{\Delta}_{\surd} \sigma$, 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 $\sigma''' \sqsupseteq^{\Delta'}_{\surd} gc(\sigma'')$ 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 $\langle (s\gamma)(t\gamma); \sigma' \rangle \Downarrow^\iota \langle w; \sigma'''' \rangle$.

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

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

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

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

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

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 $in_i\, v \in \mathcal{V}_\Delta[\![A_1 + A_2]\!]\,(n, \sigma'')$.

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

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, $\gamma \in C_\Delta[\![\Gamma]\!]\,(n, \sigma'')$ and thus $\gamma[x \mapsto v] \in C_\Delta[\![\Gamma, x : A_i]\!]\,(n, \sigma'')$. Hence, we may apply the induction hypothesis to obtain $t_i\gamma[x \mapsto v] \in \mathcal{T}_\Delta[\![B]\!]\,(n, \sigma'')$. Since all elements in the range of γ are closed terms, $t_i\gamma[x \mapsto v] = (t_i\gamma)[v/x]$ and thus $(t_i\gamma)[v/x] \in \mathcal{T}_\Delta[\![B]\!]\,(n, \sigma'')$. Consequently, $\langle (t_i\gamma)[v/x]; \sigma'' \rangle \Downarrow^\iota \langle w; \sigma''' \rangle$ with $w \in \mathcal{V}_\Delta[\![B]\!]\,(n, \sigma''')$. By definition, we thus have $\langle (case\ t\ of\ in_1\, x.t_1; in_2\, x.t_2)\gamma; \sigma' \rangle \Downarrow^\iota \langle w; \sigma''' \rangle$, as well.

- $$\frac{\Gamma, \checkmark_\theta \vdash_\Delta t : A \qquad \Gamma \vdash_\Delta \theta : Clock}{\Gamma \vdash_\Delta delay_\theta\, t : \circledcirc A}$$

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

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

thus have that

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

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

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

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 $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!]\,(n, \sigma'')$.

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

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

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

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

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

  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 $\iota(\kappa) \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma')$.

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

  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 $\sigma' = \eta_N\,\langle\kappa \mapsto w\rangle\,\eta_L$, $\vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = \big[\eta'_N\big]_\Theta$, $[\eta_L]_\Theta = \big[\eta'_L\big]_\Theta$, and $\gamma|_\Gamma \in C_\Delta[\![\Gamma]\!]\,\Big(n+1, \big(\eta'_N, \big[\eta'_L\big]_{\kappa\notin}\big)\Big)$. By induction, we thus have that $v\gamma \in \mathcal{T}_\Delta[\![\textcircled{\exists} A]\!]\,\Big(n+1, \big(\eta'_N, \big[\eta'_L\big]_{\kappa\notin}\big)\Big)$. Since $v\gamma$ is a value we have $v\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A]\!]\,\Big(n+1, \big(\eta'_N, \big[\eta'_L\big]_{\kappa\notin}\big)\Big)$ by Lemma A.5. By Lemma 5.7 we then have that

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

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

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

  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 $dom\,(\Delta)$. By definition of the context relation we also find $\kappa \in \Theta$, η_N, η_L, η'_N, η'_L such that $\sigma' = \eta_N\,\langle\kappa \mapsto w\rangle\,\eta_L$, $\vdash w : \Delta(\kappa)$, $[\eta_N]_\Theta = \big[\eta'_N\big]_\Theta$, $[\eta_L]_\Theta = \big[\eta'_L\big]_\Theta$,

and $\gamma|_\Gamma \in C_\Delta[\![\Gamma]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$. By induction hypothesis, we thus have that $v_i\gamma \in \mathcal{T}_\Delta[\![\textcircled{\exists} A_i]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$ for all $i \in \{1,2\}$. Since $v_i\gamma$ are values, we also have that $v_i\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_i]\!]\left(n+1, (\eta'_N, [\eta'_L]_{\kappa\notin})\right)$ by Lemma A.5. By Lemma 5.7 we then have that $v_i\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_i]\!]\left(n+1, (\eta_N, [\eta_L]_{\kappa\notin})\right)$ 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 $u_i \in \mathcal{V}_\Delta[\![A_i]\!](n, \sigma'')$, which by definition means that $\langle select\, v_1\gamma\, v_2\gamma; \sigma'\rangle \Downarrow^\iota \langle in_1(in_i\,(u_1,u_2)); \sigma''\rangle$ with $u_j = v_j\gamma$. It thus remains to be shown that $v_j\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n,\sigma'')$. There are two cases to consider.
  - Let $v_j\gamma = l$ for some $l \in Loc_\Delta$. From $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!]\left(n+1, (\eta_N, [\eta_L]_{\kappa\notin})\right)$ and the fact that $\kappa \notin \Theta_j$, we obtain that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!]\left(n+1, [\eta_L]_{\kappa\notin}\right)$ 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 $[\eta_L]_{\kappa\notin} \sqsubseteq^\Delta_\surd \sigma'$ and, by Lemma 5.1, $\sigma' \sqsubseteq^\Delta_\surd \sigma''$, we can then use Lemma 5.3 to conclude that $l \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n,\sigma'')$.
  - Let $v_j\gamma = wait_{\kappa'}$ for some clock κ'. But then $\Gamma \vdash_\Delta v_j : \textcircled{\exists} A_j$ is due to $\kappa' :_p A_j \in \Delta$ or $\kappa' :_{bp} A_j \in \Delta$ and thus $v_j\gamma \in \mathcal{V}_\Delta[\![\textcircled{\exists} A_j]\!](n,\sigma'')$ follows immediately by definition of the value relation.
- $\kappa \in \Theta_1 \cap \Theta_2$: By Lemma A.7, we obtain a reduction $\langle adv\, v_1\gamma); \sigma'\rangle \Downarrow^\iota \left\langle v'_1; \sigma''\right\rangle$ with $v'_1 \in \mathcal{V}_\Delta[\![A_1]\!](n,\sigma'')$, and a reduction $\langle adv\, v_2\gamma; \sigma''\rangle \Downarrow^\iota \left\langle v'_2; \sigma'''\right\rangle$ with $v'_2 \in \mathcal{V}_\Delta[\![A_2]\!](n,\sigma''')$. By definition we thus obtain a reduction $\langle select\,(v_1\gamma)\,(v_2\gamma); \sigma'\rangle \Downarrow^\iota \left\langle in_2((v'_1, v'_2)); \sigma'''\right\rangle$. 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 $in_2((v'_1, v'_2)) \in \mathcal{V}_\Delta[\![A_1 \times A_2]\!](n,\sigma''')$.

- $\dfrac{}{\Gamma \vdash_\Delta 0 : Nat}$
  $0\gamma \in \mathcal{V}_\Delta[\![Nat]\!](n,\sigma)$ follows immediately by definition.
- $\dfrac{\Gamma \vdash_\Delta t : Nat}{\Gamma \vdash_\Delta suc\, t : Nat}$
  By induction hypothesis $t\gamma \in \mathcal{T}_\Delta[\![Nat]\!](n,\sigma)$, 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, $\langle suc\, t\gamma; \sigma'\rangle \Downarrow^\iota \left\langle suc^{m+1}\, 0; \sigma''\right\rangle$ 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:

$$rec_{Nat}(s\gamma, x\ y.t\gamma, suc^k\, 0) \in \mathcal{T}_\Delta[\![A]\!](n,\sigma) \text{ for all } k \in \mathbb{N} \tag{A.1}$$

  To show that $rec_{Nat}(s\gamma, x\ y.t\gamma, u\gamma) \in \mathcal{T}_\Delta[\![A]\!](n,\sigma)$, assume some $\sigma' \sqsupseteq^\Delta_\surd \sigma$ 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 (A.1) 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 $v \in \mathcal{T}_\Delta[\![A]\!](n,\sigma''')$. 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 (A.1) 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 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

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

- $\dfrac{\Gamma^{\Box}, x : \circledcirc A \vdash_{\Delta} t : A}{\Gamma \vdash_{\Delta} fix\ x.t : A}$

We will show that

$$dfix\ x.t\gamma \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!]\,(m, \emptyset) \text{ for all } m \leq n. \tag{A.2}$$

Using (A.2), Lemma A.4, and Lemma 5.3, we then obtain that $(\gamma|_{\Gamma^{\Box}})[x \mapsto dfix\ x.t\gamma] \in C_{\Delta}[\![\Gamma^{\Box}, x : \circledcirc A]\!]\,(n, \sigma)$. Hence, by induction, $t[dfix\ x.t/x]\gamma = t(\gamma|_{\Gamma^{\Box}})[x \mapsto dfix\ x.t\gamma] \in \mathcal{T}_{\Delta}[\![A]\!]\,(n, \sigma)$, which means that we find $\langle t[dfix\ x.t/x]\gamma; \sigma' \rangle \Downarrow^{\iota} \langle v; \sigma'' \rangle$ with $v \in \mathcal{V}_{\Delta}[\![A]\!]\,(n, \sigma'')$. By definition of the machine we thus obtain the desired $\langle fix\ x.t\gamma; \sigma' \rangle \Downarrow^{\iota} \langle v; \sigma'' \rangle$.

We prove (A.2) by induction on m.

If $m = 0$, then (A.2) follows immediately from the fact that $dfix\ x.t\gamma$ is a closed term.

Let $m = m' + 1$. By Lemma 5.3 and Lemma A.4, $\gamma|_{\Gamma^{\Box}} \in C_{\Delta}[\![\Gamma^{\Box}]\!]\,(m', \emptyset)$. By the induction hypothesis (on (A.2)) we have $dfix\ x.t\gamma \in \mathcal{V}_{\Delta}[\![\circledcirc A]\!]\,(m', \emptyset)$ and thus

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

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

- $\dfrac{\Gamma \vdash_{\Delta} x : \circledcirc A}{\Gamma, \checkmark_{\theta}, \Gamma' \vdash_{\Delta} adv_{\forall}\ x : A}$

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 the definition of the context relation, Θ is well-defined and we find $\kappa \in \Theta$, $\vdash w : \Delta(\kappa)$, η_N, η_L, η'_N, η'_L such that $\sigma' = \eta_N\ \langle \kappa \mapsto w \rangle\ \eta_L$, $[\eta_N]_{\Theta} = [\eta'_N]_{\Theta}$, $[\eta_L]_{\Theta} = [\eta'_L]_{\Theta}$, and $\gamma|_{\Gamma} \in C_{\Delta'}[\![\Gamma]\!]\left(n + 1, (\eta'_N, [\eta'_L]_{\kappa \notin})\right)$. By Lemma A.3, we thus have that $\gamma(x) = dfix\ y.t$

with $dfix\ y.t \in \mathcal{V}_\Delta[\![\textcircled{\triangledown} A]\!]\left(n+1, (\eta'_N, \left[\eta'_L\right]_{\kappa\notin})\right)$. By definition, this implies that $t[dfix\ y.t/y] \in \mathcal{T}_\Delta[\![A]\!]\,(n, \emptyset)$. That is, we find a reduction $\langle t[dfix\ y.t/y]; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$ with $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma'')$, which by definition means that we also have a reduction $\langle adv_\forall\ x\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$.

- $\dfrac{\Gamma^\square \vdash_\Delta t : A}{\Gamma \vdash_\Delta box\ t : \square A}$

  We show that $box\ t\gamma \in \mathcal{V}_\Delta[\![\square A]\!]\,(n, \sigma)$. 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 $box\ t\gamma \in \mathcal{V}_\Delta[\![\square A]\!]\,(n, \sigma)$.

- $\dfrac{\Gamma \vdash_\Delta t : \square A}{\Gamma \vdash_\Delta unbox\ t : A}$

  By induction hypothesis, we have that $t\gamma \in \mathcal{T}_\Delta[\![\square A]\!]\,(n, \sigma)$. That is, $\langle t\gamma; \sigma'\rangle \Downarrow^\iota \langle box\ s; \sigma''\rangle$ for some $s \in \mathcal{T}_\Delta[\![A]\!]\,(n, \emptyset)$. Hence, $\langle s; \sigma''\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$ such that $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma''')$ which, by Lemma 5.3, implies $v \in \mathcal{V}_\Delta[\![A]\!]\,(n, \sigma''')$. Moreover, by definition of the machine we have that $\langle unbox\ t; \sigma'\rangle \Downarrow^\iota \langle v; \sigma'''\rangle$.

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

  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 $v \in \mathcal{V}_\Delta[\![A[\textcircled{\exists}(Fix\ \alpha.A)/\alpha]]\!]\,(n, \sigma'')$. Moreover, by definition of the machine we consequently have $\langle out\ t\gamma; \sigma'\rangle \Downarrow^\iota \langle v; \sigma''\rangle$.

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

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