

Modal FRP for all: Functional reactive programming without space leaks in Haskell

PATRICK BAHR 

Computer Science Department, IT University of Copenhagen, Copenhagen, Denmark
(e-mail: paba@itu.dk)

Abstract

Functional reactive programming (FRP) provides a high-level interface for implementing reactive systems in a declarative manner. However, this high-level interface has to be carefully reigned in to ensure that programs can in fact be executed in practice. Specifically, one must ensure that FRP programs are causal and can be implemented without introducing space leaks. In recent years, modal types have been demonstrated to be an effective tool to ensure these operational properties. In this paper, we present Rattus, a modal FRP language that extends and simplifies previous modal FRP calculi while still maintaining the operational guarantees for productivity, causality, and space leaks. The simplified type system makes Rattus a practical programming language that can be integrated with existing functional programming languages. To demonstrate this, we have implemented a shallow embedding of Rattus in Haskell that allows the programmer to write Rattus code in familiar Haskell syntax and seamlessly integrate it with regular Haskell code. Thus, Rattus combines the benefits enjoyed by FRP libraries such as Yampa, namely access to a rich library ecosystem (e.g., for graphics programming), with the strong operational guarantees offered by a bespoke type system. To establish the productivity, causality, and memory properties of the language, we prove type soundness using a logical relations argument fully mechanised in the Coq proof assistant.

1 Introduction

Reactive systems perform an ongoing interaction with their environment, receiving inputs from the outside, changing their internal state, and producing output. Examples of such systems include GUIs, web applications, video games, and robots. Programming such systems with traditional general-purpose imperative languages can be challenging: the components of the reactive system are put together via a complex and often confusing web of callbacks and shared mutable state. As a consequence, individual components cannot be easily understood in isolation, which makes building and maintaining reactive systems in this manner difficult and error-prone (Parent, 2006; Järvi *et al.*, 2008).

Functional reactive programming (FRP), introduced by Elliott and Hudak (1997), tries to remedy this problem by introducing time-varying values (called *behaviours* or *signals*) and *events* as a means of communication between components in a reactive system instead of shared mutable state and callbacks. Crucially, signals and events are first-class values in FRP and can be freely combined and manipulated. These high-level abstractions not

only provide a rich and expressive programming model but also make it possible for us to reason about FRP programs by simple equational methods.

Elliott and Hudak's original conception of FRP is an elegant idea that allows for direct manipulation of time-dependent data but also immediately raises the question of what the interface for signals and events should be. A naive approach would be to model discrete signals as streams defined by the following Haskell data type:¹

```
data Str a = a ::: (Str a)
```

A stream of type *Str a* thus consists of a head of type *a* and a tail of type *Str a*. The type *Str a* encodes a discrete signal of type *a*, where each element of a stream represents the value of that signal at a particular time.

Combined with the power of higher-order functional programming, we can easily manipulate and compose such signals. For example, we may apply a function to the values of a signal:

```
map :: (a → b) → Str a → Str b
map f (x ::: xs) = f x ::: map f xs
```

However, this representation is too permissive and allows the programmer to write *non-causal* programs, that is, programs where the present output depends on future input such as the following:

```
tomorrow :: Str Int → Str Int
tomorrow (x ::: xs) = xs
```

At each time step, this function takes the input of the *next* time step and returns it in the *current* time step. In practical terms, this reactive program cannot be effectively executed since we cannot compute the current value of the signal that it defines.

Much of the research in FRP has been dedicated to addressing this problem by adequately restricting the interface that the programmer can use to manipulate signals. This can be achieved by exposing only a carefully selected set of combinators to the programmer or by using a more sophisticated type system. The former approach has been very successful in practice, not least because it can be readily implemented as a library in existing languages. This library approach also immediately integrates the FRP language with a rich ecosystem of existing libraries and inherits the host language's compiler and tools. The most prominent example of this approach is Arrowised FRP (Nilsson *et al.*, 2002), as implemented in the Yampa library for Haskell (Hudak *et al.*, 2004), which takes signal functions as primitive rather than signals themselves. However, this library approach forfeits some of the simplicity and elegance of the original FRP model as it disallows direct manipulation of signals.

More recently, an alternative to this library approach has been developed (Jeffrey, 2014; Krishnaswami and Benton, 2011; Krishnaswami *et al.*, 2012; Krishnaswami, 2013; Jeltsch, 2013; Bahr *et al.*, 2019, 2021) that uses a *modal* type operator \bigcirc , which captures the notion of time. Following this idea, an element of type $\bigcirc a$ represents data of type *a* arriving in the next time step. Signals are then modelled by the type of streams defined instead as follows:

¹ Here `:::` is a data constructor written as a binary infix operator.

data $Str\ a = a :: (\bigcirc(Str\ a))$

That is, a stream of type $Str\ a$ is an element of type a now and a stream of type $Str\ a$ later, thus separating consecutive elements of the stream by one time step. Combining this modal type with guarded recursion (Nakano, 2000) in the form of a fixed point operator of type $(\bigcirc a \rightarrow a) \rightarrow a$ gives a powerful type system for reactive programming that guarantees not only causality, but also *productivity*, that is, the property that each element of a stream can be computed in finite time.

Causality and productivity of an FRP program means that it can be effectively implemented and executed. However, for practical purposes, it is also important whether it can be implemented with given finite resources. If a reactive program requires an increasing amount of memory or computation time, it will eventually run out of resources to make progress or take too long to react to input. It will grind to a halt. Since FRP programs operate on a high level of abstraction, it is typically quite difficult to reason about their space and time cost. A reactive program that exhibits a gradually slower response time, that is, its computations take longer and longer as time progresses, is said to have a *time leak*. Similarly, we say that a reactive program has a *space leak*, if its memory use is gradually increasing as time progresses, for example, if it holds on to memory while continually allocating more.

Within both lines of work – the library approach and the modal types approach – there has been an effort to devise FRP languages that avoid *implicit* space leaks. We say that a space leak is implicit if it is caused not by explicit memory allocations intended by the programmer but rather by the implementation of the FRP language holding on to old data. This is difficult to prevent in a higher-order language as closures may capture references to old data, which consequently must remain in memory for as long as the closure might be invoked. In addition, the language has to carefully balance eager and lazy evaluation: While some computations must necessarily be delayed to wait for input to arrive, we run the risk of needing to keep intermediate values in memory for too long unless we perform computations as soon as all required data have arrived. To avoid implicit space leaks, Ploeg and Claessen (2015) devised an FRP library for Haskell that avoids implicit space leaks by carefully restricting the API to manipulate events and signals. Based on the modal operator \bigcirc described above, Krishnaswami (2013) has devised a *modal* FRP calculus that permits an aggressive garbage collection strategy that rules out implicit space leaks.

Contributions. In this paper, we present Rattus, a practical modal FRP language that takes its ideas from the modal FRP calculi of Krishnaswami (2013) and Bahr *et al.* (2019, 2021) but with a simpler and less restrictive type system that makes it attractive to use in practice. Like the Simply RaTT calculus of Bahr *et al.*, we use a Fitch-style type system (Clouston, 2018), which extends typing contexts with *tokens* to avoid the syntactic overhead of the dual-context-style type system of Krishnaswami (2013). In addition, we further simplify the typing system by (1) only requiring one kind of *token* instead of two, (2) allowing tokens to be introduced without any restrictions, and (3) generalising the guarded recursion scheme. The resulting calculus is simpler and more expressive, yet still retains the operational guarantees of the earlier calculi, namely productivity, causality, and admissibility of an aggressive garbage collection strategy that prevents implicit space

leaks. We have proved these properties by a logical relations argument formalised using the Coq theorem prover (see supplementary material and Appendix B).

To demonstrate its use as a practical programming language, we have implemented Rattus as an embedded language in Haskell. This implementation consists of a library that implements the primitives of the language along with a plugin for the GHC Haskell compiler. The latter is necessary to check the more restrictive variable scope rules of Rattus and to ensure the eager evaluation strategy that is necessary to obtain the operational properties. Both components are bundled in a single Haskell library that allows the programmer to seamlessly write Rattus code alongside Haskell code. We further demonstrate the usefulness of the language with a number of case studies, including an FRP library based on streams and events as well as an arrowised FRP library in the style of Yampa. We then use both FRP libraries to implement a primitive game. The two libraries implemented in Rattus also demonstrate different approaches to FRP libraries: discrete time (streams) versus continuous time (Yampa); and first-class signals (streams) versus signal functions (Yampa). The Rattus Haskell library and all examples are included in the supplementary material.

Overview of paper. Section 2 gives an overview of the Rattus language introducing the main concepts and their intuitions. Section 3 presents a case study of a simple FRP library based on streams and events, as well as an arrowised FRP library. Section 4 presents the underlying core calculus of Rattus including its type system, its operational semantics, and our main metatheoretical results: productivity, causality, and absence of implicit space leaks. We then reflect on these results and discuss the language design of Rattus. Section 5 gives an overview of the proof of our metatheoretical results. Section 6 describes how Rattus has been implemented as an embedded language in Haskell. Section 7 reviews related work and Section 8 discusses future work.

2 Introduction to Rattus

To illustrate Rattus, we will use example programs written in the embedding of the language in Haskell. The type of streams is at the centre of these example programs:

```
data Str a = !a ::: !(⊙(Str a))
```

The annotation with bangs (!) ensures that the constructor `:::` is strict in both its arguments. We will have a closer look at the evaluation strategy of Rattus in Section 2.2.

The simplest stream one can define just repeats the same value indefinitely. Such a stream is constructed by the `constInt` function below, which takes an integer and produces a constant stream that repeats that integer at every step:

```
constInt :: Int → Str Int
constInt x = x ::: delay (constInt x)
```

Because the tail of a stream of integers must be of type $\odot(\text{Str Int})$, we have to use `delay`, which is the introduction form for the type modality \odot . Intuitively speaking, `delay` moves a computation one time step into the future. We could think of `delay` having type $a \rightarrow \odot a$, but this type is too permissive as it can cause space leaks. It would allow us to move

arbitrary computations – and the data they depend on – into the future. Instead, the typing rule for delay is formulated as follows:

$$\frac{\Gamma, \checkmark \vdash t :: A}{\Gamma \vdash \text{delay } t :: \bigcirc A}$$

This is a characteristic example of a Fitch-style typing rule (Clouston, 2018): it introduces the *token* \checkmark (pronounced ‘tick’) in the typing context Γ . A typing context consists of type assignments of the form $x :: A$, but it can also contain several occurrences of \checkmark . We can think of \checkmark as denoting the passage of one time step, that is, all variables to the left of \checkmark are one time step older than those to the right. In the above typing rule, the term t does not have access to these ‘old’ variables in Γ . There is, however, an exception: if a variable in the typing context is of a type that is time-independent, we still allow t to access them – even if the variable is one time step old. We call these time-independent types *stable* types, and in particular all base types such as *Int* and *Bool* are stable. We will discuss stable types in more detail in Section 2.1.

Formally, the variable introduction rule of Rattus reads as follows:

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

That is, if x is not of a stable type and appears to the left of a \checkmark , then it is no longer in scope.

Turning back to our definition of the *constInt* function, we can see that the recursive call *constInt* x must be of type *Str Int* in the context Γ, \checkmark , where Γ contains $x :: \text{Int}$. So x remains in scope because it is of type *Int*, which is a stable type. This would not be the case if we were to generalise *constInt* to arbitrary types:

leakyConst $:: a \rightarrow \text{Str } a$
leakyConst $x = x :: \text{delay } (\text{leakyConst } x)$ -- the rightmost occurrence of x is out of scope

In this example, x is of type a and therefore goes out of scope under delay: since a is not necessarily stable, $x :: a$ is blocked by the \checkmark introduced by delay. We can see that *leakyConst* would indeed cause a space leak by instantiating it to the type *leakyConst* $:: \text{Str Int} \rightarrow \text{Str } (\text{Str Int})$: at each time step n , it would have to store all previously observed input values from time step 0 to $n - 1$, thus making its memory usage grow linearly with time. To illustrate this on a concrete example, assume that *leakyConst* is fed the stream of numbers 0, 1, 2, . . . as input. Then, the resulting stream of type *Str (Str Int)* contains at each time step n the same stream 0, 1, 2, . . . However, the input stream arrives one integer at a time. So at time n , the input stream would have advanced to $n, n + 1, n + 2, . . .$, that is, the next input to arrive is n . Consequently, the implementation of *leakyConst* would need to have stored the previous values 0, 1, . . . $n - 1$ of the input stream.

The definition of *constInt* also illustrates the *guarded* recursion principle used in Rattus. For a recursive definition to be well-typed, all recursive calls have to occur in the presence of a \checkmark – in other words, recursive calls have to be guarded by delay. This restriction ensures that all recursive functions are productive, which means that each element of a stream can be computed in finite time. If we did not have this restriction, we could write the following obviously unproductive function:

$loop :: Str\ Int$

$loop = loop$ -- unguarded recursive call to loop is not allowed

The recursive call to $loop$ does not occur under a delay and is thus rejected by the type checker.

Let's consider an example program that transforms streams. The function inc below takes a stream of integers as input and increments each integer by 1:

$inc :: Str\ Int \rightarrow Str\ Int$

$inc\ (x :: xs) = (x + 1) :: delay\ (inc\ (adv\ xs))$

Here we have to use adv , the elimination form for \bigcirc , to convert the tail of the input stream from type $\bigcirc(Str\ Int)$ into type $Str\ Int$. Again we could think of adv having type $\bigcirc a \rightarrow a$, but this general type would allow us to write non-causal functions such as the *tomorrow* function we have seen in the introduction:

$tomorrow :: Str\ Int \rightarrow Str\ Int$

$tomorrow\ (x :: xs) = adv\ xs$ -- adv is not allowed here

This function looks one time step ahead so that the output at time n depends on the input at time $n + 1$.

To ensure causality, adv is restricted to contexts with a \checkmark :

$$\frac{\Gamma \vdash t :: \bigcirc A \quad \Gamma' \text{ tick-free}}{\Gamma, \checkmark, \Gamma' \vdash adv\ t :: A}$$

Not only does adv require a \checkmark , but it also causes all bound variables to the right of \checkmark to go out of scope. Intuitively speaking delay looks ahead one time step and adv then allows us to go back to the present. Variable bindings made in the future are therefore not accessible once we returned to the present.

Note that adv causes the variables to the right of \checkmark to go out of scope *forever*, whereas it brings variables back into scope that were previously blocked by the \checkmark . That is, variables that go out of scope due to delay can be brought back into scope by adv .

2.1 Stable types

We haven't yet made precise what stable types are. To a first approximation, types are stable if they do not contain \bigcirc or function types. Intuitively speaking, \bigcirc expresses a temporal aspect and thus types containing \bigcirc are not time-invariant. Moreover, functions can implicitly have temporal values in their closure and are therefore also excluded from stable types.

However, as a consequence, we cannot implement the map function that takes a function $f :: a \rightarrow b$ and applies it to each element of a stream of type $Str\ a$, because it would require us to apply the function f at any time in the future. We cannot do this because $a \rightarrow b$ is not a stable type (even if a and b were stable) and therefore f cannot be transported into the future. However, Rattus has the type modality \square , pronounced 'box', that turns any type A into a stable type $\square A$. Using the \square modality, we can implement map as follows:

$map :: \Box(a \rightarrow b) \rightarrow Str\ a \rightarrow Str\ b$
 $map\ f\ (x :: xs) = unbox\ f\ x :: delay\ (map\ f\ (adv\ xs))$

Instead of a function of type $a \rightarrow b$, map takes a *boxed* function f of type $\Box(a \rightarrow b)$ as its argument. That means, f is still in scope under the delay because it is of a stable type. To use f , it has to be unboxed using $unbox$, which is the elimination form for the \Box modality and simply has type $\Box a \rightarrow a$, without any restrictions.

The corresponding introduction form for \Box does come with some restrictions. It has to make sure that boxed values only refer to variables of a stable type:

$$\frac{\Gamma^\Box \vdash t :: A}{\Gamma \vdash \text{box } t :: \Box A}$$

Here, Γ^\Box denotes the typing context that is obtained from Γ by removing all variables of non-stable types and all \checkmark tokens:

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

Thus, for a well-typed term $\text{box } t$, we know that t only accesses variables of stable type.

For example, we can implement the *inc* function using map as follows:

$inc :: Str\ Int \rightarrow Str\ Int$
 $inc = map\ (\text{box } (+1))$

Using the \Box modality, we can also generalise the constant stream function to arbitrary boxed types:

$constBox :: \Box a \rightarrow Str\ a$
 $constBox\ a = unbox\ a :: delay\ (constBox\ a)$

Alternatively, we can make use of the *Stable* type class, to constrain type variables to stable types:

$const :: Stable\ a \Rightarrow a \rightarrow Str\ a$
 $const\ x = x :: delay\ (const\ x)$

Since the type of streams is not stable, the restriction to stable types disallows the instantiation of the $const$ function to the type $Str\ Int \rightarrow Str\ (Str\ Int)$, which as we have seen earlier would cause a memory leak. By contrast, $constBox$ can be instantiated to the type $\Box(Str\ Int) \rightarrow Str\ (Str\ Int)$. This is unproblematic since a value of type $\Box(Str\ Int)$ is a suspended, time-invariant computation that produces an integer stream. In other words, this computation is independent of any external input and can thus be executed at any time in the future without keeping old temporal values in memory.

So far, we have only looked at recursive definitions at the top level. Recursive definitions can also be nested, but we have to be careful how such nested recursion interacts with the typing environment. Below is an alternative definition of map that takes the boxed function f as an argument and then calls the run function that recurses over the stream:

$map :: \Box(a \rightarrow b) \rightarrow Str\ a \rightarrow Str\ b$
 $map\ f = run$

where $run :: Str\ a \rightarrow Str\ b$
 $run\ (x :: xs) = unbox\ f\ x :: delay\ (run\ (adv\ xs))$

Here, run is type-checked in a typing environment Γ that contains $f :: \Box(a \rightarrow b)$. Since run is defined by guarded recursion, we require that its definition must type-check in the typing context Γ^\Box . Because f is of a stable type, it remains in Γ^\Box and is thus in scope in the definition of run . That is, guarded recursive definitions interact with the typing environment in the same way as box , which ensures that such recursive definitions are stable and can thus safely be executed at any time in the future. As a consequence, the type checker will prevent us from writing the following leaky version of map :

$leakyMap :: (a \rightarrow b) \rightarrow Str\ a \rightarrow Str\ b$
 $leakyMap\ f = run$
where $run :: Str\ a \rightarrow Str\ b$
 $run\ (x :: xs) = f\ x :: delay\ (run\ (adv\ xs))$ -- f is no longer in scope here

The type of f is not stable, and thus it is not in scope in the definition of run .

Note that top-level defined identifiers such as map and $const$ are in scope in any context after they are defined regardless of whether there is a \checkmark or whether they are of a stable type. One can think of top-level definitions being implicitly boxed when they are defined and implicitly unboxed when they are used later on.

2.2 Operational semantics

As we have seen in the examples above, the purpose of the type modalities \bigcirc and \Box is to ensure that Rattus programs are causal, productive and without implicit space leaks. In simple terms, the latter means that temporal values, that is, values of type $\bigcirc A$, are safe to be garbage collected after two time steps. In particular, input from a stream can be safely garbage collected one time step after it has arrived. This memory property is made precise later in Section 4 along with a precise definition of the operational semantics of Rattus.

To obtain this memory property, Rattus uses an eager evaluation strategy except for $delay$ and box . That is, arguments are evaluated to values before they are passed on to functions, but special rules apply to $delay$ and box . In addition, we only allow strict data types in Rattus, which explains the use of strictness annotations in the definition of Str . This eager evaluation strategy ensures that we do not have to keep intermediate values in memory for longer than one time step.

Following the temporal interpretation of the \bigcirc modality, its introduction form $delay$ does not eagerly evaluate its argument since we may have to wait until input data arrives. For example, in the following function, we cannot evaluate $adv\ x + 1$ until the integer value of $x :: \bigcirc Int$ arrives, which is one time step from now:

$delayInc :: \bigcirc Int \rightarrow \bigcirc Int$
 $delayInc\ x = delay\ (adv\ x + 1)$

However, evaluation is only delayed by one time step, and this delay is reversed by adv . For example, $adv\ (delay\ (1 + 1))$ evaluates immediately to 2.

Turning to `box`, we can see that it needs to lazily evaluate its argument in order to maintain the memory property of `Rattus`: in the expression `box (delay 1)` of type $\Box(\bigcirc Int)$, we should not evaluate `delay 1` right away. As mention above, values of type $\bigcirc Int$ should be garbage collected after two time steps. However, boxed types are stable and can thus be moved arbitrarily into the future. Hence, by the time this boxed value is unboxed in the future, we might have already garbage collected the value of type $\bigcirc Int$ it contains.

The modal FRP calculi of Krishnaswami (2013) and Bahr *et al.* (2019, 2021) have a similar operational semantics to achieve same memory property that `Rattus` has. However, `Rattus` uses a slightly more eager evaluation strategy for `delay`: recall that `delay t` delays the computation `t` by one time step and that `adv` reverses such a delay. The operational semantics of `Rattus` reflects this intuition by first evaluating every term `t` that occurs as `delay (... adv t ...)` before evaluating `delay`. In other words, `delay (... adv t ...)` is equivalent to

$$\text{let } x = t \text{ in } \text{delay } (... \text{adv } x ...)$$

This adjustment of the operational semantics of `delay` is important, as it allows us to lift the restrictions present in previous calculi (Krishnaswami, 2013; Bahr *et al.*, 2019, 2021) that disallow guarded recursive definitions to ‘look ahead’ more than one time step. In the Fitch-style calculi of Bahr *et al.* (2019, 2021), this can be seen in the restriction to allow at most one \checkmark in the typing context. For the same reason, these two calculi also disallow function definitions in the context of a \checkmark . As a consequence, terms like `delay(delay 0)` and `delay($\lambda x.x$)` do not type-check in the calculi of Bahr *et al.* (2019, 2021).

The extension in expressive power afforded by `Rattus`’s slightly more eager evaluation strategy has immediate practical benefits: most importantly, there are no restrictions on where one can define functions. Secondly, we can write recursive functions that look several steps into the future:

```
stutter :: Int → Str Int
stutter n = n :: delay (n :: delay (stutter (n + 1)))
```

Applying `stutter` to 0 would construct a stream of numbers 0, 0, 1, 1, 2, 2, ... In order to implement `stutter` in the more restrictive language of Krishnaswami (2013) and Bahr *et al.* (2019, 2021), we would need to decompose it into two mutually recursive functions (Bahr *et al.*, 2021). A more detailed comparison of the expressive power of these calculi can be found in Section 4.5

At first glance, one might think that allowing multiple ticks could be accommodated without changing the evaluation of `delay` and instead extending the time one has to keep temporal values in memory accordingly, that is, we may safely garbage collect temporal values after $n + 1$ time steps, if we allow at most n ticks. However, as we will demonstrate in Section 4.3.3, this is not enough. Even allowing just two ticks would require us to keep temporal values in memory indefinitely, that is, it would permit implicit space leaks. On the other hand, given the more eager evaluation strategy for `delay`, we can still garbage collect all temporal values after two time steps, no matter how many ticks were involved in type-checking the program.

```

data Str a = !a ::: !( $\bigcirc$ (Str a))
map ::  $\square$ (a  $\rightarrow$  b)  $\rightarrow$  Str a  $\rightarrow$  Str b
map f (x ::: xs) = unbox f x ::: delay (map f (adv xs))
zip :: Str a  $\rightarrow$  Str b  $\rightarrow$  Str (a  $\otimes$  b)
zip (a ::: as) (b ::: bs) = (a  $\otimes$  b) ::: delay (zip (adv as) (adv bs))
scan :: Stable b  $\Rightarrow$   $\square$ (b  $\rightarrow$  a  $\rightarrow$  b)  $\rightarrow$  b  $\rightarrow$  Str a  $\rightarrow$  Str b
scan f acc (a ::: as) = acc' ::: delay (scan f acc' (adv as))
  where acc' = unbox f acc a
type Event a = Str (Maybe' a)
switch :: Str a  $\rightarrow$  Event (Str a)  $\rightarrow$  Str a
switch (x ::: xs) (Nothing'      ::: fas) = x ::: (delay switch  $\otimes$  xs  $\otimes$  fas)
switch _      (Just' (a ::: as) ::: fas) = a ::: (delay switch  $\otimes$  as  $\otimes$  fas)
switchTrans :: (Str a  $\rightarrow$  Str b)  $\rightarrow$  Event (Str a  $\rightarrow$  Str b)  $\rightarrow$  (Str a  $\rightarrow$  Str b)
switchTrans f es as = switchTrans' (f as) es as
switchTrans' :: Str b  $\rightarrow$  Event (Str a  $\rightarrow$  Str b)  $\rightarrow$  Str a  $\rightarrow$  Str b
switchTrans' (b ::: bs) (Nothing' ::: fs) as = b ::: (delay switchTrans'  $\otimes$  bs  $\otimes$  fs  $\otimes$  tail as)
switchTrans' _      (Just' f      ::: fs) as = b' ::: (delay switchTrans'  $\otimes$  bs'  $\otimes$  fs'  $\otimes$  tail as)
  where (b' ::: bs') = f as

```

Fig. 1. Small library for streams and events.

3 Reactive programming in Rattus

In this section, we showcase how Rattus can be used for reactive programming. To this end, we implement a small library of combinators for programming with streams and events. We then use this library to implement a simple game. Finally, we implement a Yampa-style library and re-implement the game using that library instead. The full sources of both implementations of the game are included in the supplementary material along with a third variant that uses a combinator library based on monadic streams (Perez *et al.*, 2016).

3.1 Programming with streams and events

To illustrate how Rattus facilitates working with streams and events, we have implemented a small set of combinators, as shown in Figure 1. The *map* function should be familiar by now. The *zip* function combines two streams similarly to Haskell's *zip* function defined on lists. Note however that instead of the normal pair type, we use a strict pair type:

```
data a  $\otimes$  b = !a  $\otimes$  !b
```

It is like the normal pair type (a, b) , but when constructing a strict pair $s \otimes t$, the two components s and t are evaluated to weak head normal form.

The *scan* function is similar to Haskell's *scanl* function on lists: given a stream of values v_0, v_1, v_2, \dots , the expression *scan* (*box* *f*) *v* computes the stream

$$f \ v \ v_0, \quad f \ (f \ v \ v_0) \ v_1, \quad f \ (f \ (f \ v \ v_0) \ v_1) \ v_2, \quad \dots$$

If one would want a variant of *scan* that is closer to Haskell's *scanl*, that is, the result starts with the value v instead of $f \ v \ v_0$, one can simply replace the first occurrence of *acc*' in the

definition of *scan* with *acc*. Note that the type *b* has to be stable in the definition of *scan* so that $acc' :: b$ is still in scope under delay.

A central component of FRP is that it must provide a way to react to events. In particular, it must support the ability to *switch* behaviour in response to the occurrence of an event. There are different ways to represent events. The simplest representation defines events of type *a* as streams of type *Maybe a*. However, we will use the strict variant of the *Maybe* type:

data *Maybe'* *a* = *Just'* !*a* | *Nothing'*

We can then devise a simple *switch* combinator that reacts to events. Given an initial stream *xs* and an event *e* that may produce a stream, *switch xs e* initially behaves as *xs* but changes to the new stream provided by the occurrence of an event. In this implementation, the behaviour changes *every time* an event occurs, not only the first time. For a one-shot variant of *switch*, we would just have to change the second equation to

switch _ (*Just'* *as* :: _) = *as*

In the definition of *switch*, we use the applicative operator \otimes defined as follows:

$(\otimes) :: \bigcirc(a \rightarrow b) \rightarrow \bigcirc a \rightarrow \bigcirc b$
 $f \otimes x = \text{delay } ((\text{adv } f) (\text{adv } x))$

Instead of using \otimes , we could have also written $\text{delay } (\text{switch } (\text{adv } xs) (\text{adv } fas))$ instead.

Finally, *switchTrans* is a variant of *switch* that switches to a new stream function rather than just a stream. It is implemented using the variant *switchTrans'*, which takes just a stream as its first argument instead of a stream function.

3.2 A simple reactive program

To put our bare-bones FRP library to use, let's implement a simple single-player variant of the classic game Pong: the player has to move a paddle at the bottom of the screen to bounce a ball and prevent it from falling.² The core behaviour is described by the following stream function:

pong :: *Str Input* → *Str (Pos* \otimes *Float)*
pong inp = *zip ball pad where*
pad :: *Str Float*
pad = *padPos inp*
ball :: *Str Pos*
ball = *ballPos (zip pad inp)*

It receives a stream of inputs (button presses and how much time has passed since the last input) and produces a stream of pairs consisting of the 2D position of the ball and the *x* coordinate of the paddle. Its implementation uses two helper functions to compute these two components. The position of the paddle only depends on the input, whereas the position of the ball also depends on the position of the paddle (since it may bounce off it):

² So it is rather like Breakout, but without the bricks.

```

padPos :: Str (Input) → Str Float
padPos = map (box fst') ∘ scan (box padStep) (0 ⊗ 0)
padStep :: (Float ⊗ Float) → Input → (Float ⊗ Float)
padStep (pos ⊗ vel) inp = ...

ballPos :: Str (Float ⊗ Input) → Str Pos
ballPos = map (box fst') ∘ scan (box ballStep) ((0 ⊗ 0) ⊗ (20 ⊗ 50))
ballStep :: (Pos ⊗ Vel) → (Float ⊗ Input) → (Pos ⊗ Vel)
ballStep (pos ⊗ vel) (pad ⊗ inp) = ...

```

Both auxiliary functions follow the same structure. They use a *scan* to compute the position and the velocity of the object, while consuming the input stream. The velocity is only needed to compute the position and is therefore projected away afterward using *map*. Here, *fst'* is the first projection for the strict pair type. We can see that the ball starts at the centre of the screen (at coordinates (0, 0)) and moves toward the upper right corner (with velocity (20, 50)).

This simple game only requires a static dataflow network. To demonstrate the ability to express dynamically changing dataflow networks in Rattus, we briefly discuss three refinements of the pong game, each of which introduces different kinds of dynamic behaviours. In each case, we reuse the existing stream functions that describe the basic behaviour of the ball and the paddle, but we combine them using different combinators. Thus, these examples also demonstrate the modularity that Rattus affords.

First refinement. We change the implementation of *pong* so that it allows the player to reset the game, for example, after the ball has fallen off the screen:

```

pong1 :: Str Input → Str (Pos ⊗ Float)
pong1 inp = zip ball pad where
  pad = padPos inp
  ball = switchTrans ballPos
           (map (box ballTrig) inp) -- starting ball behaviour
           (zip pad inp)           -- trigger restart on pressing reset button
           -- input to the switch

ballTrig :: Input → Maybe' (Str (Float ⊗ Input) → Str Pos)
ballTrig inp = if reset inp then Just' ballPos else Nothing'

```

To achieve this behaviour, we use the *switchTrans* combinator, which we initialise with the original behaviour of the ball. The event that will trigger the switch is constructed by mapping *ballTrig* over the input stream, which will create an event of type *Event (Str (Float ⊗ Input) → Str Pos)*, which will be triggered every time the player hits the reset button.

Second refinement.³ We can further refine this behaviour of the game so that it automatically resets once the ball has fallen off the screen. This requires a feedback loop since the behaviour of the ball depends on the current position of the ball. Such a feedback loop can

³ See `Haskell-library/examples/game` directory in the supplementary material.

be constructed using a variant of the *switchTrans* combinator that takes a delayed event as argument:

$$dswitchTrans :: (Str\ a \rightarrow Str\ b) \rightarrow \bigcirc(Event\ (Str\ a \rightarrow Str\ b)) \rightarrow (Str\ a \rightarrow Str\ b)$$

Thus, the event we pass on to *dswitchTrans* can be constructed from the output of *dswitchTrans* by using guarded recursion, which closes the feedback loop:

$$pong_2 :: Str\ Input \rightarrow Str\ (Pos \otimes Float)$$

$$pong_2\ inp = zip\ ball\ pad\ \mathbf{where}$$

$$pad = padPos\ inp$$

$$ball = dswitchTrans\ ballPos$$

$$(\text{delay}\ (\text{map}\ (\text{box}\ ballTrig')\ (pong_2\ (\text{adv}\ (tl\ inp))))))$$

$$(\text{zip}\ pad\ inp)$$

$$ballTrig' :: Pos \otimes Float \rightarrow Maybe'\ (Str\ (Float \otimes Input) \rightarrow Str\ Pos)$$

$$ballTrig'\ ((_ \otimes y) \otimes _) = \mathbf{if}\ y < bottomScreenEdge\ \mathbf{then}\ Just'\ ballPos\ \mathbf{else}\ Nothing'$$

Final refinement.⁴ Finally, we change the game a bit so as to allow the player to spawn new balls at any time and remove balls that are currently in play. To this end, we introduce a type *ObjAction a b* that represents events that may remove or add objects defined by a stream function of type *Str a → Str b*:

$$\mathbf{data}\ ObjAction\ a\ b = Remove\ | Add\ !(Str\ a \rightarrow Str\ b)$$

To keep it simple, we only have two operations: we may add an object by giving its stream function, or we may remove the oldest object. Given the strict list type

$$\mathbf{data}\ List\ a = Nil\ | Cons\ !a\ !(List\ a)$$

we can implement a combinator that can react to such events:

$$objTrans :: Event\ (ObjAction\ a\ b) \rightarrow List\ (Str\ b) \rightarrow Str\ a \rightarrow Str\ (List\ b)$$

This combinator takes three arguments: an event that may trigger operations to manipulate the list of active objects, the initial list of active objects, and the input stream. The result is a stream that at each step contains a list with the current value of each active object.

To use this combinator, we revise the *ballTrig* function so that it produces events to add or remove balls in response to input by the player, and we initialise the list of objects as the empty list:

$$pong_3 :: Str\ Input \rightarrow Str\ (List\ Pos \otimes Float)$$

$$pong_3\ inp = zip\ balls\ pad\ \mathbf{where}$$

$$pad :: Str\ Float$$

$$pad = padPos\ inp$$

$$balls :: Str\ (List\ Pos)$$

$$balls = objTrans\ (\text{map}\ (\text{box}\ ballTrig'')\ inp)\ Nil\ (\text{zip}\ pad\ inp)$$

$$ballTrig'' :: Input \rightarrow Maybe'\ (ObjAction\ (Float \otimes Input)\ Pos)$$

$$ballTrig''\ inp\ | addBall\ inp = Just'\ (Add\ ballPos)$$

⁴ See `Haskell-library/examples/game-multi` directory in the supplementary material.

$$\begin{aligned} | \text{removeBall } \text{inp} &= \text{Just' Remove} \\ | \text{otherwise} &= \text{Nothing'} \end{aligned}$$

Instead of a stream of type *Str Pos* to describe a single ball, we now have a stream of type *Str (List Pos)* to describe multiple balls.

The *objTrans* combinator demonstrates that Rattus can accommodate dataflow networks that dynamically grow and shrink. But more realistic variants of this combinator can be implemented as well. For example, by replacing *List* with a *Map* data structure, objects may have identifiers and can be removed or manipulated based on that identifier. Orthogonal to that, we may also allow objects to destroy themselves and spawn new objects. This can be achieved by describing objects using trees instead of streams:

data *Tree a* = *End* | *Next !a !(*○(*Tree a*)*)* | *Branch !a !(*○(*Tree a*)*) !(*○(*Tree a*)*)*

A variant of *objTrans* can then be implemented where the type *Str b* in *objTrans* and *ObjAction a b* is replaced by *Tree b*. For example, this can be used in the pong game to make balls destroy themselves once they disappear from screen or make a ball split into two balls when it hits a certain surface.

3.3 Arrowised FRP

The benefit of a modal FRP language is that we can directly interact with signals and events in a way that guarantees causality. A popular alternative to ensure causality is arrowised FRP (Nilsson *et al.*, 2002), which takes *signal functions* as primitive and uses Haskell's arrow notation (Paterson, 2001) to construct them. By implementing an arrowised FRP library in Rattus instead of plain Haskell, we can not only guarantee causality but also productivity and the absence of implicit space leaks. As we will see, this forces us to slightly restrict the API of arrowised FRP compared to Yampa. Furthermore, this exercise demonstrates that Rattus can also be used to implement a continuous-time FRP library, in contrast to the discrete-time FRP library from Section 3.1.

At the centre of arrowised FRP is the *Arrow* type class shown in Figure 2. If we can implement a signal function type *SF a b* that implements the *Arrow* class, we can benefit from the convenient notation Haskell provides for it. For example, assuming we have signal functions *ballPos* :: *SF (Float* ⊗ *Input) Pos* and *padPos* :: *SF Input Float* describing the positions of the ball and the paddle from our game in Section 3.2, we can combine these as follows:

```
pong :: SF Input (Pos ⊗ Float)
pong = proc inp → do pad ← padPos ← inp
      ball ← ballPos ←(pad ⊗ inp)
      returnA ←(ball ⊗ pad)
```

The Rattus definition of *SF* is almost identical to the original Haskell definition from Nilsson *et al.* (2002). The only difference is the use of strict types and the insertion of the ○ modality to make it a guarded recursive type:

data *SF a b* = *SF !(Float* → *a* → (○(*SF a b*) ⊗ *b*))

class <i>Category</i> $a \Rightarrow \text{Arrow } a$ where <i>arr</i> :: $(b \rightarrow c) \rightarrow a\ b\ c$ <i>first</i> :: $a\ b\ c \rightarrow a\ (b, d)\ (c, d)$ <i>second</i> :: $a\ b\ c \rightarrow a\ (d, b)\ (d, c)$ $(**)$:: $a\ b\ c \rightarrow a\ b'\ c' \rightarrow a\ (b, b')\ (c, c')$ $(\&\&\&)$:: $a\ b\ c \rightarrow a\ b\ c' \rightarrow a\ b\ (c, c')$	class <i>Category</i> cat where <i>id</i> :: $cat\ a\ a$ (\circ) :: $cat\ b\ c \rightarrow cat\ a\ b \rightarrow cat\ a\ c$
	class <i>Arrow</i> $a \Rightarrow \text{ArrowLoop } a$ where <i>loop</i> :: $a\ (b, d)\ (c, d) \rightarrow a\ b\ c$

Fig. 2. Arrow type class.

This implements a continuous-time signal function using sampling, where the additional argument of type *Float* indicates the time passed since the previous sample.

Implementing the methods of the *Arrow* type class is straightforward with the exception of the *arr* method. In fact, we cannot implement *arr* in Rattus at all. Because the first argument is not stable, it falls out of scope in the recursive call:

```
arr :: (a → b) → SF a b
arr f = SF (λ_ a → (delay (arr f), f a))  -- f is not in scope under delay
```

The situation is similar to the *map* function, and we must box the function argument so that it remains available at all times in the future:

```
arrBox :: □(a → b) → SF a b
arrBox f = SF (λ_ a → (delay (arrBox f), unbox f a))
```

That is, the *arr* method could be a potential source for space leaks. However, in practice, *arr* does not seem to cause space leaks and thus its use in conventional arrowised FRP libraries should be safe. Nonetheless, in Rattus, we have to replace *arr* with the more restrictive variant *arrBox*. But fortunately, this does not prevent us from using the arrow notation: Rattus treats *arr f* as a short hand for *arrBox* (*box f*), which allows us to use the arrow notation while making sure that *box f* is well-typed, that is, *f* only refers to variables of stable type.

The *Arrow* type class only provides a basic interface for constructing *static* signal functions. To permit dynamic behaviour, we need to provide additional combinators, for example, for switching signals and for recursive definitions. The *rSwitch* combinator corresponds to the *switchTrans* combinator from Figure 1:

```
rSwitch :: SF a b → SF (a ⊗ Maybe' (SF a b)) b
```

This combinator allows us to implement our game so that it resets to its start position if we hit the reset button:⁵

```
pong1 :: SF Input (Pos ⊗ Float)
pong1 = proc inp → do pad ← padPos ← inp
      let event = if reset inp then Just' ballPos else Nothing'
      ball ← rSwitch ballPos ← ((pad ⊗ inp) ⊗ event)
      returnA ← (ball ⊗ pad)
```

Arrows can be endowed with a very general recursion principle by instantiating the *loop* method in the *ArrowLoop* type class shown in Figure 2. However, *loop* cannot

⁵ See `Haskell-library/examples/game-arrow` directory in the supplementary material.

be implemented in Rattus as it would break the productivity property. Moreover, *loop* depends crucially on lazy evaluation and is thus a source for space leaks.

Instead of *loop*, we implement a different recursion principle that corresponds to guarded recursion:

$$\text{loopPre} :: d \rightarrow SF (b \otimes d) (c \otimes \bigcirc d) \rightarrow SF b c$$

Intuitively speaking, this combinator constructs a signal function from b to c with the help of an internal state of type d . The first argument initialises the state, and the second argument is a signal function that turns input of type b into output of type c while also updating the internal state. Apart from the addition of the \bigcirc modality and strict pair types, this definition has the same type as Yampa's *loopPre*. Alternatively, we could drop the \bigcirc modality and constrain d to be stable. The use of *loopPre* instead of *loop* introduces a delay by one sampling step (as indicated by the \bigcirc modality) and thus ensures productivity. However, in practice, such a delay can be avoided by refactoring the underlying signal function.

Using the *loopPre* combinator, we can implement the signal function of the ball:

$$\text{ballPos} :: SF (\text{Float} \otimes \text{Input}) \text{Pos}$$

$$\text{ballPos} = \text{loopPre} (20 \otimes 50) \text{run where}$$

$$\text{run} :: SF ((\text{Float} \otimes \text{Input}) \otimes \text{Vel}) (\text{Pos} \otimes \bigcirc \text{Vel})$$

$$\text{run} = \mathbf{proc} ((\text{pad} \otimes \text{inp}) \otimes v) \rightarrow \mathbf{do} p \leftarrow \text{integral} (0 \otimes 0) \prec v$$

$$\text{returnA} \prec (p \otimes \text{delay} (\text{calcNewVelocity} \text{pad} p v))$$

Here, we also use the *integral* combinator that computes the integral of a signal using a simple approximation that sums up rectangles under the curve:

$$\text{integral} :: (\text{Stable } a, \text{VectorSpace } a \text{ } s) \Rightarrow a \rightarrow SF a a$$

The signal function for the paddle can be implemented in a similar fashion. The complete code of the case studies presented in this section can be found in the supplementary material.

4 Core calculus

In this section, we present the core calculus of Rattus, which we call $\lambda_{\mathcal{R}}$. The purpose of $\lambda_{\mathcal{R}}$ is to formally present the language's Fitch-style typing rules, its operational semantics, and to formally prove the central operational properties, that is, productivity, causality, and absence of implicit space leaks. To this end, the calculus is stripped down to its essence: a simply typed lambda calculus extended with guarded recursive types $\text{Fix } \alpha. A$, a guarded fixed point combinator, and the two type modalities \square and \bigcirc . Since general inductive types and polymorphic types are orthogonal to the issue of operational properties in reactive programming, we have omitted these for the sake of clarity.

4.1 Type system

Figure 3 defines the syntax of $\lambda_{\mathcal{R}}$. Besides guarded recursive types, guarded fixed points, and the two type modalities, we include standard sum and product types along with unit

Types	$A, B ::= \alpha \mid 1 \mid \text{Int} \mid A \times B \mid A + B \mid A \rightarrow B \mid \Box A \mid \bigcirc A \mid \text{Fix } \alpha.A$
Stable types	$S, S' ::= 1 \mid \text{Int} \mid \Box A \mid S \times S' \mid S + S'$
Values	$v, w ::= \langle \rangle \mid \bar{n} \mid \lambda x.t \mid \langle v, w \rangle \mid \text{in}_i v \mid \text{box } t \mid \text{into } v \mid \text{fix } x.t \mid l$
Terms	$s, t ::= \langle \rangle \mid \bar{n} \mid \lambda x.t \mid \langle s, t \rangle \mid \text{in}_i t \mid \text{box } t \mid \text{into } t \mid \text{fix } x.t \mid l$ $\mid x \mid t_1 t_2 \mid t_1 + t_2 \mid \text{adv } t \mid \text{delay } t \mid \text{unbox } t \mid \text{out } t$ $\mid \pi_i \mid \text{case } t \text{ of } \text{in}_1 x.t_1; \text{in}_2 x.t_2 \mid \text{let } x = s \text{ in } t$

Fig. 3. Syntax of (stable) types, terms, and values. In typing rules, only closed types are considered (i.e., each occurrence of a type variable α is in the scope of a $\text{Fix } \alpha$).

$$\frac{}{\emptyset \vdash_{\checkmark}} \qquad \frac{\Gamma \vdash_{\checkmark}}{\Gamma, x : A \vdash_{\checkmark}} \qquad \frac{\Gamma \vdash_{\checkmark}}{\Gamma, \checkmark \vdash_{\checkmark}}$$

Fig. 4. Well-formed contexts.

and integer types. The type $\text{Str } A$ of streams of type A is represented as the guarded recursive type $\text{Fix } \alpha.A \times \alpha$. Note the absence of \bigcirc in this type. When unfolding guarded recursive types such as $\text{Fix } \alpha.A \times \alpha$, the \bigcirc modality is inserted implicitly: $\text{Fix } \alpha.A \times \alpha \cong A \times \bigcirc(\text{Fix } \alpha.A \times \alpha)$. This ensures that guarded recursive types are by construction always guarded by the \bigcirc modality.

For the sake of the operational semantics, the syntax also includes heap locations l . However, as we shall see, heap locations cannot be used by the programmer directly, because there is no typing rule for them. Instead, heap locations are allocated and returned by delay in the operational semantics. This is a standard approach for languages with references (see e.g., Abramsky *et al.* (1998); Krishnaswami (2013)).

Typing contexts, defined in Figure 4, consist of variable typings $x : A$ and \checkmark tokens. If a typing context contains no \checkmark , we call it *tick-free*. The complete set of typing rules for λ_{\checkmark} is given in Figure 5. The typing rules for Rattus presented in Section 2 appear in the same form also here, except for replacing Haskell’s $::$ operator with the more standard notation. The remaining typing rules are entirely standard, except for the typing rule for the guarded fixed point combinator fix .

The typing rule for fix follows Nakano’s fixed point combinator and ensures that the calculus is productive. In addition, following Krishnaswami (2013), the rule enforces the body t of the fixed point to be stable by strengthening the typing context to Γ^{\square} . Moreover, we follow Bahr *et al.* (2021) and assume x to be of type $\Box(\bigcirc A)$ instead of $\bigcirc A$. As a consequence, recursive calls may occur at any time in the future, that is, not necessarily in the very next time step. In conjunction with the more general typing rule for delay , this allows us to write recursive function definitions that, like *stutter* in Section 2.2, look several steps into the future.

To see how the recursion syntax of Rattus translates into the fixed point combinator of λ_{\checkmark} , let us reconsider the *constInt* function:

$\text{constInt} :: \text{Int} \rightarrow \text{Str Int}$
 $\text{constInt } x = x ::: \text{delay } (\text{constInt } x)$

For readability of the corresponding λ_{\checkmark} term, we use the shorthand $s ::: t$ for the λ_{\checkmark} term into $\langle s, t \rangle$. Recall that $\text{Str } A$ is represented as $\text{Fix } \alpha.A \times \alpha$ in λ_{\checkmark} . That is, given $s : A$ and

$$\begin{array}{c}
\frac{\Gamma, x : A, \Gamma' \vdash_{\mathcal{W}} \quad \Gamma' \text{ tick-free or } A \text{ stable}}{\Gamma, x : A, \Gamma' \vdash_{\mathcal{W}} x : A} \quad \frac{\Gamma \vdash_{\mathcal{W}}}{\Gamma \vdash_{\mathcal{W}} \langle \rangle : 1} \quad \frac{n \in \mathbb{Z}}{\Gamma \vdash_{\mathcal{W}} \bar{n} : \text{Int}} \\
\frac{\Gamma \vdash_{\mathcal{W}} s : \text{Int} \quad \Gamma \vdash_{\mathcal{W}} t : \text{Int}}{\Gamma \vdash_{\mathcal{W}} s + t : \text{Int}} \quad \frac{\Gamma, x : A \vdash_{\mathcal{W}} t : B}{\Gamma \vdash_{\mathcal{W}} \lambda x. t : A \rightarrow B} \quad \frac{\Gamma \vdash_{\mathcal{W}} s : A \quad \Gamma, x : A \vdash_{\mathcal{W}} t : B}{\Gamma \vdash_{\mathcal{W}} \text{let } x = s \text{ in } t : B} \\
\frac{\Gamma \vdash_{\mathcal{W}} t : A \rightarrow B \quad \Gamma \vdash_{\mathcal{W}} t' : A}{\Gamma \vdash_{\mathcal{W}} t t' : B} \quad \frac{\Gamma \vdash_{\mathcal{W}} t : A \quad \Gamma \vdash_{\mathcal{W}} t' : B}{\Gamma \vdash_{\mathcal{W}} \langle t, t' \rangle : A \times B} \\
\frac{\Gamma \vdash_{\mathcal{W}} t : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash_{\mathcal{W}} \pi_i t : A_i} \quad \frac{\Gamma \vdash_{\mathcal{W}} t : A_i \quad i \in \{1, 2\}}{\Gamma \vdash_{\mathcal{W}} \text{in}_i t : A_1 + A_2} \\
\frac{\Gamma, x : A_i \vdash_{\mathcal{W}} t_i : B \quad \Gamma \vdash_{\mathcal{W}} t : A_1 + A_2 \quad i \in \{1, 2\}}{\Gamma \vdash_{\mathcal{W}} \text{case } t \text{ of } \text{in}_1 x. t_1; \text{in}_2 x. t_2 : B} \quad \frac{\Gamma, \checkmark \vdash_{\mathcal{W}} t : A}{\Gamma \vdash_{\mathcal{W}} \text{delay } t : \bigcirc A} \\
\frac{\Gamma \vdash_{\mathcal{W}} t : \bigcirc A \quad \Gamma' \text{ tick-free}}{\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{adv } t : A} \quad \frac{\Gamma \vdash_{\mathcal{W}} t : \square A}{\Gamma \vdash_{\mathcal{W}} \text{unbox } t : A} \quad \frac{\Gamma^{\square} \vdash_{\mathcal{W}} t : A}{\Gamma \vdash_{\mathcal{W}} \text{box } t : \square A} \\
\frac{\Gamma \vdash_{\mathcal{W}} t : A[\bigcirc(\text{Fix } \alpha. A)/\alpha]}{\Gamma \vdash_{\mathcal{W}} \text{into } t : \text{Fix } \alpha. A} \quad \frac{\Gamma \vdash_{\mathcal{W}} t : \text{Fix } \alpha. A}{\Gamma \vdash_{\mathcal{W}} \text{out } t : A[\bigcirc(\text{Fix } \alpha. A)/\alpha]} \\
\frac{\Gamma^{\square}, x : \square(\bigcirc A) \vdash_{\mathcal{W}} t : A}{\Gamma \vdash_{\mathcal{W}} \text{fix } x. t : A}
\end{array}$$

Fig. 5. Typing rules.

$t : \bigcirc(\text{Str } A)$, we have that $s ::: t$ is of type $\text{Str } A$. Using this notation, the above definition translates into the following $\lambda_{\mathcal{W}}$ term:

$$\text{constInt} = \text{fix } r. \lambda x. x ::: \text{delay}(\text{adv}(\text{unbox } r)x)$$

The recursive notation is simply translated into a fixed point $\text{fix } r. t$ where the recursive occurrence of constInt is replaced by $\text{adv}(\text{unbox } r)$. The variable r is of type $\square(\bigcirc(\text{Int} \rightarrow \text{Str } \text{Int}))$ and applying unbox followed by adv turns it into type $\text{Int} \rightarrow \text{Str } \text{Int}$. Moreover, the restriction that recursive calls must occur in a context with \checkmark makes sure that this transformation from recursion notation to fixed point combinator results in a well-typed $\lambda_{\mathcal{W}}$ term.

The typing rule for $\text{fix } x. t$ also explains the treatment of recursive definitions that are nested inside a top-level definition. The typing context Γ is turned into Γ^{\square} when type-checking the body t of the fixed point.

For example, reconsider the following ill-typed definition of leakyMap :

$$\text{leakyMap} :: (a \rightarrow b) \rightarrow \text{Str } a \rightarrow \text{Str } b$$

$$\text{leakyMap } f = \text{run}$$

$$\text{where } \text{run} :: \text{Str } a \rightarrow \text{Str } b$$

$$\text{run } (x ::: xs) = f x ::: \text{delay}(\text{leakyMap}(\text{adv } xs))$$

Translated into λ_{\checkmark} , the definition looks like this:

$$\begin{aligned} leakyMap &= \lambda f. \text{fix } r. \lambda s. \text{let } x = \text{head } s \text{ in let } xs = \text{tail } s \\ &\quad \text{in } f \ x :: \text{delay}((\text{adv } (\text{unbox } r)) (\text{adv } xs)) \end{aligned}$$

The pattern matching syntax is translated into projection functions `head` and `tail` that decompose a stream into its head and tail, respectively, that is,

$$\text{head} = \lambda x. \pi_1 (\text{out } x) \quad \text{tail} = \lambda x. \pi_2 (\text{out } x)$$

More importantly, the variable f bound by the outer lambda abstraction is of a function type and thus not stable. Therefore, it is not in scope in the body of the fixed point.

4.2 Operational semantics

The operational semantics is given in two steps: to execute a λ_{\checkmark} program, it is first translated into a more restrictive variant of λ_{\checkmark} , which we call λ_{\checkmark} . The resulting λ_{\checkmark} program is then executed using an abstract machine that ensures the absence of implicit space leaks by construction. By presenting the operational semantics in two stages, we avoid the more complicated set-up of an abstract machine that is capable of directly executing λ_{\checkmark} programs. As we show in the example in Section 4.3.3, such a machine would need to perform some restricted form of partial evaluation under delay and under lambda abstractions.

4.2.1 Translation to λ_{\checkmark}

The λ_{\checkmark} calculus has the same syntax as λ_{\checkmark} , but the former has a more restrictive type system. It restricts typing contexts to contain at most one \checkmark and restricts the typing rules for lambda abstraction and delay as follows:

$$\frac{|\Gamma|, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \lambda x. t : A \rightarrow B} \quad \frac{|\Gamma|, \checkmark \vdash_{\checkmark} t : A}{\Gamma \vdash_{\checkmark} \text{delay } t : \bigcirc A} \quad \begin{array}{l} |\Gamma| = \Gamma \quad \text{if } \Gamma \text{ is tick-free} \\ |\Gamma, \checkmark, \Gamma'| = \Gamma^{\square}, \Gamma' \end{array}$$

The construction $|\Gamma|$ turns Γ into a tick-free context, which ensures that we have at most one \checkmark – even for nested occurrences of delay – and that the body of a lambda abstraction is not in the scope of a \checkmark . All other typing rules are the same as for λ_{\checkmark} . The λ_{\checkmark} calculus is a fragment of the λ_{\checkmark} calculus in the sense that $\Gamma \vdash_{\checkmark} t : A$ implies $\Gamma \vdash_{\checkmark} t : A$.

Any closed λ_{\checkmark} term can be translated into a corresponding λ_{\checkmark} term by exhaustively applying the following rewrite rules:

$$\begin{aligned} \text{delay}(K[\text{adv } t]) &\longrightarrow \text{let } x = t \text{ in delay}(K[\text{adv } x]) \quad \text{if } t \text{ is not a variable} \\ \lambda x. K[\text{adv } t] &\longrightarrow \text{let } y = \text{adv } t \text{ in } \lambda x. (K[y]) \end{aligned}$$

where K is a term with a single hole that does not occur in the scope of `delay`, `adv`, `box`, `fix`, or lambda abstraction; and $K[t]$ is the term obtained from K by replacing its hole with the term t .

For example, consider the closed λ_{\checkmark} term $\lambda f. \text{delay}(\lambda x. \text{adv } (\text{unbox } f) (x + 1))$ of type $\square \bigcirc (\text{Int} \rightarrow \text{Int}) \rightarrow \bigcirc (\text{Int} \rightarrow \text{Int})$. Applying the second rewrite rule followed by the first

rewrite rule, this term rewrites to a λ_{\checkmark} term of the same type as follows:

$$\begin{aligned} & \lambda f.\text{delay}(\lambda x.\text{adv}(\text{unbox } f)(x + 1)) \\ \longrightarrow & \lambda f.\text{delay}(\text{let } y = \text{adv}(\text{unbox } f) \text{ in } \lambda x.y(x + 1)) \\ \longrightarrow & \lambda f.\text{let } z = \text{unbox } f \text{ in } \text{delay}(\text{let } y = \text{adv } z \text{ in } \lambda x.y(x + 1)) \end{aligned}$$

In a well-typed λ_{\checkmark} term, each subterm $\text{adv } t$ must occur in the scope of a corresponding delay. The above rewrite rules make sure that the subterm t is evaluated before the delay. This corresponds to the intuition that delay moves ahead in time and adv moves back in time – thus the two cancel out one another.

One can show that the rewrite rules are strongly normalising and type-preserving (in λ_{\checkmark}). Moreover, any closed term in λ_{\checkmark} that cannot be further rewritten is also well-typed in λ_{\checkmark} . As a consequence, we can exhaustively apply the rewrite rules to a closed term of λ_{\checkmark} to transform it into a closed term of λ_{\checkmark} :

Theorem 4.1. *For each $\vdash_{\checkmark} t : A$, we can effectively construct a term t' with $t \longrightarrow^* t'$ and $\vdash_{\checkmark} t' : A$.*

Below we give a brief overview of the three components of the proof of Theorem 4.1. For the full proof, we refer the reader to Appendix A.

Strong normalisation. To show that \longrightarrow is strongly normalising, we define for each term t a natural number $d(t)$ such that, whenever $t \longrightarrow t'$, then $d(t) > d(t')$. We define $d(t)$ to be the sum of the depth of all redex occurrences in t (i.e., subterms that match the left-hand side of a rewrite rule). Since each rewrite step $t \longrightarrow t'$ removes a redex or replaces a redex with a new redex at a strictly smaller depth, we have that $d(t) > d(t')$.

Subject reduction. We want to prove that $\Gamma \vdash_{\checkmark} s : A$ and $s \longrightarrow t$ implies $\Gamma \vdash_{\checkmark} t : A$. To this end, we proceed by induction on $s \longrightarrow t$. In case the reduction $s \longrightarrow t$ is due to congruence closure, $\Gamma \vdash_{\checkmark} t : A$ follows immediately by the induction hypothesis. Otherwise, s matches the left-hand side of one of the rewrite rules. Each of these two cases follows from the induction hypothesis and one of the following two properties:

- (1) If $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} K[\text{adv } t] : A$ and Γ' tick-free, then $\Gamma \vdash_{\checkmark} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\checkmark} K[\text{adv } x] : A$ for some B .
- (2) If $\Gamma, \Gamma' \vdash_{\checkmark} K[\text{adv } t] : A$ and Γ' tick-free, then $\Gamma \vdash_{\checkmark} \text{adv } t : B$ and $\Gamma, x : B, \Gamma' \vdash_{\checkmark} K[x] : A$ for some B .

Both properties can be proved by a straightforward induction on K . The proofs rely on the fact that due to the typing rule for adv , we know that if $\Gamma, \Gamma' \vdash_{\checkmark} K[\text{adv } t] : A$ for a tick-free Γ' , then all of t 's free variables must be in Γ .

Exhaustiveness. Finally, we need to show that $\vdash_{\checkmark} t : A$ with $t \not\rightarrow$ implies $\vdash_{\checkmark} t : A$, that is, if we cannot rewrite t any further it must be typable in λ_{\checkmark} as well. In order to prove this property by induction, we must generalise it to open terms: if $\Gamma \vdash_{\checkmark} t : A$ for a context Γ with at most one tick and $t \not\rightarrow$, then $\Gamma \vdash_{\checkmark} t : A$. We prove this implication by induction on

t and a case distinction on the last typing rule in the derivation of $\Gamma \vdash_{\checkmark} t : A$. For almost all cases, $\Gamma \vdash_{\checkmark} t : A$ follows from the induction hypothesis since we find a corresponding typing rule in λ_{\checkmark} that is either the same as in λ_{\checkmark} or has a side condition is satisfied by our assumption that Γ have at most one tick. We are thus left with two interesting cases: the typing rules for delay and lambda abstraction, given that Γ contains exactly one tick (the zero-tick cases are trivial). Each of these two cases follows from the induction hypothesis and one of the following two properties:

- (1) If $\Gamma_1, \checkmark, \Gamma_2 \vdash_{\checkmark} t : A$, Γ_2 contains a tick, and $\text{delay } t \rightarrow$, then $\Gamma_1^{\square}, \Gamma_2 \vdash_{\checkmark} t : A$.
- (2) If $\Gamma_1, \checkmark, \Gamma_2 \vdash_{\checkmark} t : A$ and $\lambda x.t \rightarrow$, then $\Gamma_1^{\square}, \Gamma_2 \vdash_{\checkmark} t : A$.

Both properties can be proved by a straightforward induction on the typing derivation. The proof of (1) uses the fact that t cannot have nested occurrences of adv and thus any occurrence of adv only needs the tick that is already present in Γ_2 . In turn, (2) holds due to the fact that all occurrences of adv in t must be guarded by an occurrence of delay in t itself, and thus the tick between Γ_1 and Γ_2 is not needed. Note that (2) is about λ_{\checkmark} as we first apply the induction hypothesis and then apply (2).

4.2.2 Abstract machine for λ_{\checkmark}

To prove the absence of implicit space leaks, we devise an abstract machine that after each time step deletes all data from the previous time step. That means, the operational semantics is *by construction* free of implicit space leaks. This approach, pioneered by Krishnaswami (2013), allows us to reduce the proof of no implicit space leaks to a proof of type soundness.

At the centre of this approach is the idea to execute programs in a machine that has access to a store consisting of up to two separate heaps: a ‘now’ heap from which we can retrieve delayed computations, and a ‘later’ heap where we must store computations that should be performed in the next time step. Once the machine advances to the next time step, it will delete the ‘now’ heap and the ‘later’ heap will become the new ‘now’ heap.

The machine consists of two components: the *evaluation semantics*, presented in Figure 6, which describes the operational behaviour of λ_{\checkmark} within a single time step; and the *step semantics*, presented in Figure 7, which describes the behaviour of a program over time, for example, how it consumes and constructs streams.

The evaluation semantics is given as a deterministic big-step operational semantics, where we write $\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$ to indicate that starting with the store σ , the term t evaluates to the value v and the new store σ' . A store σ can be of one of two forms: either it consists of a single heap η_L , that is, $\sigma = \eta_L$, or it consists of two heaps η_N and η_L , written $\sigma = \eta_N \checkmark \eta_L$. The ‘later’ heap η_L contains delayed computations that may be retrieved and executed in the next time step, whereas the ‘now’ heap η_N contains delayed computations from the previous time step that can be retrieved and executed now. We can only write to η_L and only read from η_N . However, when one time step passes, the ‘now’ heap η_N is deleted and the ‘later’ heap η_L becomes the new ‘now’ heap. This shifting of time is part of the step semantics in Figure 7, which we turn to shortly.

Heaps are simply finite mappings from *heap locations* to terms. To allocate fresh heap locations, we assume a function $\text{alloc } (\cdot)$ that takes a store σ of the form η_L or $\eta_N \checkmark \eta_L$ and

$$\begin{array}{c}
\frac{}{\langle v; \sigma \rangle \Downarrow \langle v; \sigma \rangle} \quad \frac{\langle t; \sigma \rangle \Downarrow \langle \bar{m}; \sigma' \rangle \quad \langle t'; \sigma' \rangle \Downarrow \langle \bar{n}; \sigma'' \rangle}{\langle t + t'; \sigma \rangle \Downarrow \langle \bar{m} + \bar{n}; \sigma'' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle u; \sigma' \rangle \quad \langle t'; \sigma' \rangle \Downarrow \langle u'; \sigma'' \rangle}{\langle \langle t, t' \rangle; \sigma \rangle \Downarrow \langle \langle u, u' \rangle; \sigma'' \rangle} \quad \frac{\langle t; \sigma \rangle \Downarrow \langle \langle v_1, v_2 \rangle; \sigma' \rangle \quad i \in \{1, 2\}}{\langle \pi_i(t); \sigma \rangle \Downarrow \langle v_i; \sigma' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle \quad i \in \{1, 2\}}{\langle \text{in}_i(t); \sigma \rangle \Downarrow \langle \text{in}_i(v); \sigma' \rangle} \quad \frac{\langle s; \sigma \rangle \Downarrow \langle v; \sigma' \rangle \quad \langle t[v/x]; \sigma' \rangle \Downarrow \langle v'; \sigma'' \rangle}{\langle \text{let } x = s \text{ in } t; \sigma \rangle \Downarrow \langle v'; \sigma'' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle \text{in}_i(u); \sigma' \rangle \quad \langle t_i[v/x]; \sigma' \rangle \Downarrow \langle u_i; \sigma'' \rangle \quad i \in \{1, 2\}}{\langle \text{case } t \text{ of in}_1 x.t_1; \text{in}_2 x.t_2; \sigma \rangle \Downarrow \langle u_i; \sigma'' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle \lambda x.s; \sigma' \rangle \quad \langle t'; \sigma' \rangle \Downarrow \langle v; \sigma'' \rangle \quad \langle s[v/x]; \sigma'' \rangle \Downarrow \langle v'; \sigma''' \rangle}{\langle t t'; \sigma \rangle \Downarrow \langle v'; \sigma''' \rangle} \\
\frac{l = \text{alloc } (\sigma)}{\langle \text{delay } t; \sigma \rangle \Downarrow \langle l; \sigma, l \mapsto t \rangle} \quad \frac{\langle t; \eta_N \rangle \Downarrow \langle l; \eta'_N \rangle \quad \langle \eta'_N(l); \eta'_N \checkmark \eta_L \rangle \Downarrow \langle v; \sigma' \rangle}{\langle \text{adv } t; \eta_N \checkmark \eta_L \rangle \Downarrow \langle v; \sigma' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle \text{box } t'; \sigma' \rangle \quad \langle t'; \sigma' \rangle \Downarrow \langle v; \sigma'' \rangle}{\langle \text{unbox } t; \sigma \rangle \Downarrow \langle v; \sigma'' \rangle} \quad \frac{\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle}{\langle \text{into } t; \sigma \rangle \Downarrow \langle \text{into } v; \sigma' \rangle} \\
\frac{\langle t; \sigma \rangle \Downarrow \langle \text{into } v; \sigma' \rangle}{\langle \text{out } t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle} \quad \frac{\langle t[\text{box } (\text{delay } (\text{fix } x.t))/x]; \sigma \rangle \Downarrow \langle v; \sigma' \rangle}{\langle \text{fix } x.t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle}
\end{array}$$

Fig. 6. Evaluation semantics.

$$\frac{\langle t; \eta \checkmark \rangle \Downarrow \langle v ::: l; \eta_N \checkmark \eta_L \rangle}{\langle t; \eta \rangle \xrightarrow{v} \langle \text{adv } l; \eta_L \rangle} \quad \frac{\langle t; \eta, l^* \mapsto v ::: l^* \checkmark l^* \mapsto \langle \rangle \rangle \Downarrow \langle v' ::: l; \eta_N \checkmark \eta_L, l^* \mapsto \langle \rangle \rangle}{\langle t; \eta \rangle \xrightarrow{v/v'} \langle \text{adv } l; \eta_L \rangle}$$

Fig. 7. Step semantics for streams.

returns a heap location l that is not in the domain of η_L . Given such a fresh heap location l and a term t , we write $\sigma, l \mapsto t$ to denote the store η'_L or $\eta_N \checkmark \eta'_L$, respectively, where $\eta'_L = \eta_L, l \mapsto t$, that is, η'_L is obtained from η_L by extending it with a new mapping $l \mapsto t$.

Applying `delay` to a term t stores t in a fresh location l on the ‘later’ heap and then returns l . Conversely, if we apply `adv` to such a delayed computation, we retrieve the term from the ‘now’ heap and evaluate it.

4.3 Main results

In this section, we present the main metatheoretical results. Namely, that the core calculus λ_{\checkmark} enjoys the desired productivity, causality, and memory properties (see Theorem 4.2 and Theorem 4.3 below). The proof of these results is sketched in Section 5 and is fully mechanised in the accompanying Coq proofs. In order to formulate and prove these

metatheoretical results, we devise a step semantics that describes the behaviour of reactive programs. Here, we consider two kinds of reactive programs: terms of type $Str A$ and terms of type $Str A \rightarrow Str B$. The former just produce infinite streams of values of type A , whereas the latter are reactive processes that produce a value of type B for each input value of type A . The purpose of the step semantics is to formulate clear metatheoretical properties and to subsequently prove them using the fundamental property of our logical relation (Theorem 5.1). In principle, we could formulate similar step semantics for the Yampa-style signal functions from Section 3.3 or other basic FRP types such as resumptions (Krishnaswami, 2013) and then derive similar metatheoretical results.

4.3.1 Productivity of streams

The step semantics \xRightarrow{v} from Figure 7 describes the unfolding of streams of type $Str A$. Given a closed term $\vdash_{\checkmark} t : Str A$, it produces an infinite reduction sequence

$$\langle t; \emptyset \rangle \xRightarrow{v_0} \langle t_1; \eta_1 \rangle \xRightarrow{v_1} \langle t_2; \eta_2 \rangle \xRightarrow{v_2} \dots$$

where \emptyset denotes the empty heap and each v_i has type A . In each step, we have a term t_i and the corresponding heap η_i of delayed computations. According to the definition of the step semantics, we evaluate $\langle t_i; \eta_i \checkmark \rangle \Downarrow \langle v_i :: l; \eta'_i \checkmark \eta_{i+1} \rangle$, where η'_i is η_i but possibly extended with some additional delayed computations and η_{i+1} is the new heap with delayed computations for the next time step. Crucially, the old heap η'_i is thrown away. That is, by construction, old data is not implicitly retained but garbage collected immediately after we completed the current time step.

To see this garbage collection strategy in action, consider the following definition of the stream of consecutive numbers starting from some given number:

```
from :: Int → Str Int
from n = n :: delay (from (n + 1))
```

This definition translates to the λ_{\checkmark} term $\text{fix } r.\lambda n.n :: \text{delay}(\text{adv}(\text{unbox } r)(n + \bar{1}))$, which, in turn, rewrites into the following λ_{\checkmark} term:

$$from = \text{fix } r.\lambda n.n :: \text{let } r' = \text{unbox } r \text{ in } \text{delay}(\text{adv } r'(n + \bar{1}))$$

Let's see how the term $from \bar{0}$ of type $Str Int$ is executed on the machine:

$$\begin{aligned} \langle from \bar{0}; \emptyset \rangle &\xRightarrow{\bar{0}} \langle \text{adv } l'_1; l_1 \mapsto from, l'_1 \mapsto \text{adv } l_1 (\bar{0} + \bar{1}) \rangle \\ &\xRightarrow{\bar{1}} \langle \text{adv } l'_2; l_2 \mapsto from, l'_2 \mapsto \text{adv } l_2 (\bar{1} + \bar{1}) \rangle \\ &\xRightarrow{\bar{2}} \langle \text{adv } l'_3; l_3 \mapsto from, l'_3 \mapsto \text{adv } l_3 (\bar{2} + \bar{1}) \rangle \\ &\vdots \end{aligned}$$

In each step, the heap contains at location l_i the fixed point $from$ and at location l'_i the delayed computation produced by the occurrence of delay in the body of the fixed point. The old versions of the delayed computations are garbage collected after each step and only the most recent version survives.

Our main result is that the execution of λ_{\checkmark} terms by the machine described in Figure 6 and 7 is safe. To describe the type of the produced values precisely, we need to restrict

ourselves to streams over types whose evaluation is not suspended, which excludes function and modal types. This idea is expressed in the notion of *value types*, defined by the following grammar:

$$\text{Value types } V, W ::= 1 \mid \text{Int} \mid U \times W \mid U + W$$

We can then prove the following theorem, which both expresses the fact that the aggressive garbage collection strategy of Rattus is safe, and that stream programs are productive:

Theorem 4.2 (productivity of streams). *Given a term $\vdash_{\checkmark} t : \text{Str } A$ with A a value type, there is an infinite reduction sequence*

$$\langle t; \emptyset \rangle \xRightarrow{v_0} \langle t_1; \eta_1 \rangle \xRightarrow{v_1} \langle t_2; \eta_2 \rangle \xRightarrow{v_2} \dots \quad \text{such that } \vdash_{\checkmark} v_i : A \text{ for all } i \geq 0.$$

The restriction to value types is only necessary for showing that each output value v_i has the correct type.

4.3.2 Productivity and causality of stream transducers

The step semantics $\xRightarrow{v/v'}$ from Figure 7 describes how a term of type $\text{Str } A \rightarrow \text{Str } B$ transforms a stream of inputs into a stream of outputs in a step-by-step fashion. Given a closed term $\vdash_{\checkmark} t : \text{Str } A \rightarrow \text{Str } B$, and an infinite stream of input values $\vdash_{\checkmark} v_i : A$, it produces an infinite reduction sequence

$$\langle t(\text{adv } l^*); \emptyset \rangle \xRightarrow{v_0/v'_0} \langle t_1; \eta_1 \rangle \xRightarrow{v_1/v'_1} \langle t_2; \eta_2 \rangle \xRightarrow{v_2/v'_2} \dots$$

where each output value v'_i has type B .

The definition of $\xRightarrow{v/v'}$ assumes that we have some fixed heap location l^* , which acts both as interface to the currently available input value and as a stand-in for future inputs that are not yet available. As we can see above, this stand-in value l^* is passed on to the stream function in the form of the argument $\text{adv } l^*$. Then, in each step, we evaluate the current term t_i in the current heap η_i :

$$\langle t_i; \eta_i, l^* \mapsto v_i \text{ :: } l^* \checkmark l^* \mapsto \langle \rangle \rangle \Downarrow \langle v'_i \text{ :: } l; \eta'_i \checkmark \eta_{i+1}, l^* \mapsto \langle \rangle \rangle$$

which produces the output v'_i and the new heap η_{i+1} . Again the old heap η'_i is simply dropped. In the ‘later’ heap, the operational semantics maps l^* to the placeholder value $\langle \rangle$, which is safe since the machine never reads from the ‘later’ heap. Then in the next reduction step, we replace that placeholder value with $v_{i+1} \text{ :: } l^*$ which contains the newly received input value v_{i+1} .

For an example, consider the following function that takes a stream of integers and produces the stream of prefix sums:

$\text{sum} :: \text{Str } \text{Int} \rightarrow \text{Str } \text{Int}$

$\text{sum} = \text{run } 0$ **where**

$\text{run} :: \text{Int} \rightarrow \text{Str } \text{Int} \rightarrow \text{Str } \text{Int}$

$\text{run } \text{acc} (x \text{ :: } xs) = \text{let } \text{acc}' = \text{acc} + x$

in $\text{acc}' \text{ :: } \text{delay } (\text{run } \text{acc}' (\text{adv } xs))$

This function definition translates to the following term *sum* in the λ_{\checkmark} calculus:

$$\begin{aligned} run &= \text{fix } r.\lambda acc.\lambda s. \text{ let } x = \text{head } s \text{ in let } xs = \text{tail } s \text{ in let } acc' = acc + x \text{ in} \\ &\quad \text{let } r' = \text{unbox } r \text{ in } acc' ::: \text{delay}(\text{adv } r' acc' (\text{adv } xs)) \\ sum &= run \bar{0} \end{aligned}$$

Let's look at the first three steps of executing the *sum* function with 2, 11 and 5 as its first three input values:

$$\begin{aligned} &\langle sum(\text{adv } l^*); \emptyset \rangle \\ \xrightarrow{\bar{2}/\bar{2}} &\langle \text{adv } l'_1; l_1 \mapsto run, l'_1 \mapsto \text{adv } l_1 \bar{2}(\text{adv } l^*) \rangle \\ \xrightarrow{\bar{11}/\bar{13}} &\langle \text{adv } l'_2; l_2 \mapsto run, l'_2 \mapsto \text{adv } l_2 \bar{13}(\text{adv } l^*) \rangle \\ \xrightarrow{\bar{5}/\bar{18}} &\langle \text{adv } l'_3; l_3 \mapsto run, l'_3 \mapsto \text{adv } l_3 \bar{18}(\text{adv } l^*) \rangle \\ &\vdots \end{aligned}$$

In each step of the computation, l_i stores the fixed point *run* and l'_i stores the computation that calls that fixed point with the new accumulator value (2, 13 and 18, respectively) and the tail of the current input stream.

We can prove the following theorem, which again expresses the fact that the garbage collection strategy of Rattus is safe, and that stream processing functions are both productive and causal:

Theorem 4.3 (causality and productivity of stream transducers). *Given a term $\vdash_{\checkmark} t : \text{Str } A \rightarrow \text{Str } B$ with B a value type, and an infinite sequence of values $\vdash_{\checkmark} v_i : A$, there is an infinite reduction sequence*

$$\langle t(\text{adv } l^*); \emptyset \rangle \xrightarrow{v_0/v'_0} \langle t_1; \eta_1 \rangle \xrightarrow{v_1/v'_1} \langle t_2; \eta_2 \rangle \xrightarrow{v_2/v'_2} \dots \quad \text{such that } \vdash_{\checkmark} v'_i : B \text{ for all } i \geq 0.$$

Since the operational semantics is deterministic, in each step $\langle t_i; \eta_i \rangle \xrightarrow{v_i/v'_i} \langle t_{i+1}; \eta_{i+1} \rangle$, the resulting output v'_{i+1} and new state of the computation $\langle t_{i+1}; \eta_{i+1} \rangle$ are uniquely determined by the previous state $\langle t_i; \eta_i \rangle$ and the input v_i . Thus, v'_i and $\langle t_{i+1}; \eta_{i+1} \rangle$ are independent of future inputs v_j with $j > i$.

4.3.3 Space leak in the naive operational semantics

Theorems 4.2 and 4.3 show that λ_{\checkmark} terms can be executed without implicit space leaks after they have been translated into λ_{\checkmark} terms. To demonstrate the need of the translation step, we give an example that illustrates what would happen if we would skip it.

Since both λ_{\checkmark} and λ_{\checkmark} share the same syntax, the abstract machine from Section 4.2.2 could in principle be used for λ_{\checkmark} terms directly, without transforming them to λ_{\checkmark} first. However, we can construct a well-typed λ_{\checkmark} term for which the machine will try to dereference a heap location that has previously been garbage collected. We might conjecture that we could accommodate direct execution of λ_{\checkmark} by increasing the number of available heaps in the abstract machine so that it matches the number of ticks that were necessary to type-check the term we wish to execute. But this is not the case: we can construct a λ_{\checkmark} term that can be type-checked with only two ticks but requires an unbounded number

of heaps to safely execute directly. In other words, programs in λ_{\checkmark} may exhibit implicit space leaks, if we just run them using the evaluation strategy of the abstract machine from Section 4.2.2.

Such implicit space leaks can occur in λ_{\checkmark} for two reasons: (1) a lambda abstraction that appears in the scope of a \checkmark , and (2) a term that requires more than one \checkmark to type-check. Bahr *et al.* (2019) give an example of (1), which translates to λ_{\checkmark} . The following term of type $\bigcirc(\text{Str Int}) \rightarrow \bigcirc(\text{Str Int})$ is an example of (2):

$$t = \text{fix } r. \lambda x. \text{delay}(\text{head}(\text{adv } x) :: \text{adv}(\text{unbox } r)(\text{delay}(\text{adv}(\text{tail}(\text{adv } x))))))$$

The abstract machine would get stuck trying to dereference a heap location that was previously garbage collected. The problem is that the second occurrence of x is nested below two occurrences of delay . As a consequence, when $\text{adv } x$ is evaluated, the heap location bound to x is two time steps old and has been garbage collected already.

To see this behaviour on a concrete example, recall the λ_{\checkmark} term $\text{from} : \text{Int} \rightarrow \text{Str Int}$ from Section 4.3.1, which produces a stream of consecutive integers. Using t and from , we can construct the λ_{\checkmark} term $\bar{0} :: t(\text{delay}(\text{from } \bar{1}))$ of type Str Int , which we can run on the abstract machine:⁶

$$\langle \bar{0} :: t(\text{delay}(\text{from } \bar{1})); \emptyset \rangle \xRightarrow{\bar{0}} \langle \text{adv } l_2; \eta_1 \rangle \xRightarrow{\bar{1}} \langle \text{adv } l_4; \eta_2 \rangle \rightleftharpoons$$

$$\text{where } \eta_1 = l_1 \mapsto \text{from } \bar{1},$$

$$l_2 \mapsto \text{head}(\text{adv } l_1) :: \text{adv}(\text{unbox}(\text{box}(\text{delay } t))(\text{delay}(\text{adv}(\text{tail}(\text{adv } l_1))))))$$

$$\eta_2 = l_3 \mapsto \text{adv}(\text{tail}(\text{adv } l_1)),$$

$$l_4 \mapsto \text{head}(\text{adv } l_3) :: \text{adv}(\text{unbox}(\text{box}(\text{delay } t))(\text{delay}(\text{adv}(\text{tail}(\text{adv } l_3))))))$$

In the last step, the machine would get stuck trying to evaluate $\text{adv } l_4$, since this in turn requires evaluating $\text{head}(\text{adv } l_3)$ and $\text{adv}(\text{tail}(\text{adv } l_1))$, which would require looking up l_1 in η_1 , which has already been garbage collected. Assuming we modified the machine so that it does not perform garbage collection, but instead holds on to the previous heaps η_1, η_2 , it would continue as follows:

$$\langle \text{adv } l_2; \eta_1 \rangle \xRightarrow{\bar{1}} \langle \text{adv } l_4; \eta_1 \checkmark \eta_2 \rangle \xRightarrow{\bar{2}} \langle \text{adv } l_6; \eta_1 \checkmark \eta_2 \checkmark \eta_3 \rangle$$

$$\text{where } \eta_3 = l_5 \mapsto \text{adv}(\text{tail}(\text{adv } l_3)),$$

$$l_6 \mapsto \text{head}(\text{adv } l_5) :: \text{adv}(\text{unbox}(\text{box}(\text{delay } t))(\text{delay}(\text{adv}(\text{tail}(\text{adv } l_5))))))$$

The next execution step would require the machine to look up l_6 , which in turn requires l_5 , l_3 , and eventually l_1 . We could continue this arbitrarily far into the future, and in each step the machine would need to look up l_1 in the very first heap η_1 .

This examples suggests that the cause of this space leak is the fact that the machine leaves the nested terms $\text{adv } l_1, \text{adv } l_3$, etc. unevaluated on the heap. As our metatheoretical results in this section show, the translation into λ_{\checkmark} avoids this problem. Indeed, if we apply the rewrite rules from Section 4.2.1 to t , we obtain the λ_{\checkmark} term $\bar{0} :: t'(\text{delay}(\text{from } \bar{1}))$, where

⁶ To help make the examples in this section more readable, we elide heap locations if they are not referenced anywhere in the term or other parts of the heap.

$$t' = \text{fix } r. \lambda x. \text{let } r' = \text{unbox } r \\ \text{in } \text{delay}(\text{head}(\text{adv } x) ::: \text{adv } r'(\text{let } y = \text{tail}(\text{adv } x) \text{ in } \text{delay}(\text{adv } y)))$$

This term can then be safely executed on the abstract machine:

$$\langle \bar{0} ::: t' (\text{delay } (\text{from } \bar{1}); \emptyset) \rangle \xrightarrow{\bar{0}} \langle \text{adv } l_3; \eta_1 \rangle \xrightarrow{\bar{1}} \langle \text{adv } l_8; \eta_2 \rangle \xrightarrow{\bar{2}} \dots$$

where $\eta_1 = l_1 \mapsto \text{from } \bar{1}, l_2 \mapsto t'$

$$l_3 \mapsto \text{head}(\text{adv } l_1) ::: \text{adv } l_2 (\text{let } y = \text{tail}(\text{adv } l_1) \text{ in } \text{delay}(\text{adv } y))$$

$$\eta_2 = l_4 \mapsto \text{from}, l_5 \mapsto \text{adv } l_4 (\bar{1} + \bar{1}), l_6 \mapsto \text{adv } l_5, l_7 \mapsto t'$$

$$l_8 \mapsto \text{head}(\text{adv } l_6) ::: \text{adv } l_7 (\text{let } y = \text{tail}(\text{adv } l_6) \text{ in } \text{delay}(\text{adv } y))$$

4.4 Limitations

Now that we have formally precise statements about the operational properties of Rattus' core calculus, we should make sure that we understand what they mean in practice and what their limitations are. In simple terms, the productivity and causality properties established by Theorems 4.2 and 4.3 state that reactive programs in Rattus can be executed effectively – they always make progress and never depend on data that is not yet available. However, Rattus allows calling general Haskell functions, for which we can make no such operational guarantees. This trade-off is intentional as we wish to make Haskell's rich library ecosystem available to Rattus. Similar trade-offs are common in foreign function interfaces that allow function calls into another language. For instance, Haskell code may call C functions.

In addition, by virtue of the operational semantics, Theorems 4.2 and 4.3 also imply that programs can be executed without implicitly retaining memory – thus avoiding *implicit space leaks*. This follows from the fact that in each step, the step semantics (cf. Figure 7) discards the 'now' heap and only retains the 'later' heap for the next step. However, similarly to the calculi of Krishnaswami (2013) and Bahr *et al.* (2021, 2019), Rattus still allows *explicit space leaks* (we may still construct data structures to hold on to an increasing amount of memory) as well as *time leaks* (computations may take an increasing amount of time). Below we give some examples of these behaviours.

Given a strict list type *List* (cf. Section 3.2), we can construct a function that buffers the entire history of its input stream:

$$\text{buffer} :: \text{Stable } a \Rightarrow \text{Str } a \rightarrow \text{Str } (\text{List } a) \\ \text{buffer} = \text{scan } (\text{box } (\lambda xs \ x \rightarrow \text{Cons } x \ xs)) \ \text{Nil}$$

Given that we have a function $\text{sum} :: \text{List } \text{Int} \rightarrow \text{Int}$ that computes the sum of a list of numbers, we can write the following alternative implementation of the *sums* function using *buffer*:

$$\text{leakySums1} :: \text{Str } \text{Int} \rightarrow \text{Str } \text{Int} \\ \text{leakySums1} = \text{map } (\text{box } \text{sum}) \circ \text{buffer}$$

At each time step, this function adds the current input integer to the buffer of type *List Int* and then computes the sum of the current value of that buffer. This function exhibits both

a space leak (buffering a steadily growing list of numbers) and a time leak (the time to compute each element of the resulting stream increases at each step). However, these leaks are explicit in the program.

An example of a time leak is found in the following alternative implementation of the *sums* function:

```
leakySums2 :: Str Int → Str Int
leakySums2 (x :: xs) = x :: delay (map (box (+x)) (leakySums2 (adv xs)))
```

In each step, we add the current input value x to each future output. The closure $(+x)$, which is Haskell shorthand notation for $\lambda y \rightarrow y + x$, stores each input value x .

None of the above space and time leaks are prevented by Rattus. The space leaks in *buffer* and *leakySums1* are explicit, since the desire to buffer the input is explicitly stated in the program. That is, while Rattus prevents implicit space leaks, it still allows programmers to allocate memory as they see fit. The other example is more subtle as the leaky behaviour is rooted in a time leak as the program constructs an increasing computation in each step. This shows that the programmer still has to be careful about time leaks. Note that these leaky functions can also be implemented in the calculi of Krishnaswami (2013) and Bahr *et al.* (2019, 2021), although some reformulation is necessary for the Simply RaTT calculus of Bahr *et al.* (2019).

4.5 Language design considerations

The design of Rattus and its core calculus is derived from the calculi of Krishnaswami (2013) and Bahr *et al.* (2019, 2021), which are the only other modal FRP calculi with a garbage collection result similar to ours. In the following, we review the differences of Rattus compared to these calculi with the aim of illustrating how the design of Rattus follows from our goal to simplify previous calculi while still maintaining their strong operational properties.

Like the present work, the Simply RaTT calculus of Bahr *et al.* (2019) uses a Fitch-style type system, which provides lighter syntax to interact with the \square and \bigcirc modality compared to Krishnaswami's use of qualifiers in his calculus. In addition, Simply RaTT also dispenses with the allocation tokens of Krishnaswami's calculus by making the *box* primitive call-by-name. By contrast, Krishnaswami's calculus is closely related to dual-context systems and requires the use of pattern matching as elimination forms of the modalities. The Lively RaTT calculus of Bahr *et al.* (2021) extends Simply RaTT with temporal inductive types to express liveness properties. But otherwise, Lively RaTT is similar to Simply RaTT and the discussion below equally applies to Lively RaTT as well.

As discussed in Section 2.2, Simply RaTT restricts where ticks may occur, which disallows terms like $\text{delay}(\text{delay } 0)$ and $\text{delay}(\lambda x.x)$. In addition, Simply RaTT has a more complicated typing rule for guarded fixed points (cf. Figure 8). In addition to \checkmark , Simply RaTT uses the token \sharp to serve the role that stabilisation of a context Γ to Γ^\square serves in Rattus. But Simply RaTT only allows one such \sharp token, and \checkmark may only appear to the right of \sharp . Moreover, fixed points in Simply RaTT produce terms of type $\square A$ rather than just A . Taken together, this makes the syntax and typing for guarded recursive function

$$\frac{\Gamma, \sharp, x : \Box A \vdash t : A}{\Gamma \vdash \text{fix } x.t : \Box A} \quad \frac{\Gamma \vdash t : \Box A \quad \Gamma' \text{ token-free}}{\Gamma, \sharp, \Gamma' \vdash \text{unbox } t : A} \quad \frac{\Gamma, x : A \vdash t : B \quad \Gamma' \text{ tick-free}}{\Gamma \vdash \lambda x.t : A \rightarrow B}$$

Fig. 8. Selected typing rules from Bahr *et al.* (2019).

definitions more complicated and less intuitive. For example, the *map* function would be defined as follows in Simply RaTT:

$$\begin{aligned} \text{map} &: \Box(a \rightarrow b) \rightarrow \Box(\text{Str } a \rightarrow \text{Str } b) \\ \text{map } f \# (a :: as) &= \text{unbox } f \ a :: (\text{map } f \ \otimes \ as) \end{aligned}$$

Here, # is used to indicate that the argument *f* is to the left of the # token and only because of the presence of this token can we use the unbox combinator on *f* (cf. Figure 8). Additionally, the typing of recursive definitions is somewhat confusing: *map* has return type $\Box(\text{Str } a \rightarrow \text{Str } b)$ but when used in a recursive call as seen above, *map f* is of type $\Box(\text{Str } a \rightarrow \text{Str } b)$ instead. Moreover, we cannot call *map* recursively on its own. All recursive calls must be of the form *map f*, the exact pattern that appears to the left of the #. This last restriction rules out the following variant of *map* that is meant to take two functions and alternately apply them to a stream:

$$\begin{aligned} \text{alterMap} &: \Box(a \rightarrow b) \rightarrow \Box(a \rightarrow b) \rightarrow \Box(\text{Str } a \rightarrow \text{Str } b) \\ \text{alterMap } f \ g \# (a :: as) &= \text{unbox } f \ a :: (\text{alterMap } g \ f \ \otimes \ as) \end{aligned}$$

Only *alterMap f g* is allowed as recursive call, but not *alterMap g f*. By contrast, *alterMap* can be implemented in Rattus without problems:

$$\begin{aligned} \text{alterMap} &: \Box(a \rightarrow b) \rightarrow \Box(a \rightarrow b) \rightarrow \text{Str } a \rightarrow \text{Str } b \\ \text{alterMap } f \ g \ (a :: as) &= f \ a :: (\text{delay } (\text{alterMap } g \ f) \ \otimes \ as) \end{aligned}$$

In addition, because guarded recursive functions always have a boxed return type, definitions in Simply RaTT are often littered with calls to unbox. For example, the function *pong₁* from Section 3.2 would be implemented as follows in Simply RaTT:

$$\begin{aligned} \text{pong}_1 &:: \Box(\text{Str } \text{Input} \rightarrow \text{Str } (\text{Pos} \otimes \text{Float})) \\ \text{pong}_1 \# \text{inp} &= \text{unbox } \text{zip } \text{ball } \text{pad} \ \mathbf{where} \\ \text{pad} &= \text{unbox } \text{padPos } \text{inp} \\ \text{ball} &= \text{unbox } \text{switchTrans } (\text{unbox } \text{ballPos } \text{inp}) \\ &\quad (\text{unbox } (\text{triggerMap } \text{ballTrig } \text{inp}) \\ &\quad (\text{unbox } \text{zip } \text{pad } \text{inp}) \end{aligned}$$

By making the type of guarded fixed points *A* rather than $\Box A$, we avoid all of the above issues related to guarded recursive definitions. Moreover, the unrestricted unbox combinator found in Rattus follows directly from this change in the guarded fixed point typing. If we were to change the typing rule for fix in Simply RaTT so that fix has type *A* instead of $\Box A$, we would be able to define an unrestricted unbox combinator $\lambda x.\text{fix } y.\text{unbox } x$ of type $\Box A \rightarrow A$.

Conversely, if we keep the unbox combinator of Simply RaTT but lift some of the restrictions regarding the # token such as allowing \checkmark not only to the right of a # or allowing more than one # token, then we would break the garbage collection property and thus

permit leaky programs. In such a modified version of Simply RaTT, we would be able to type-check the following term:

$$\Gamma \vdash (\lambda x. \text{delay}(\text{fix } y. \text{unbox}(\text{adv } x) ::: y))(\text{delay}(\text{box } 0)) : \bigcirc(\square \text{StrNat})$$

where $\Gamma = \sharp$ if we were to allow two \sharp tokens or Γ empty, if we were to allow the \checkmark to occur left of the \sharp . The above term stores the value `box 0` on the heap and then constructs a stream, which at each step tries to read this value from the heap location. Hence, in order to maintain the garbage collection property in Rattus, we had to change the typing rule for `unbox`.

In addition, Rattus permits recursive functions that look more than one time step into the future (e.g., *stutter* from Section 2.2), which is not possible in Krishnaswami (2013) and Bahr *et al.* (2019, 2021). However, we conjecture that Krishnaswami's calculus can be adapted to allow this by changing the typing rule for `fix` in a similar way as we did for Rattus.

We should note that that the $\#$ token found in Simply RaTT has some benefit over the Γ^\square construction. It allows the calculus to reject some programs with time leaks, for example, *leakySums2* from Section 4.4, because of the condition in the rule for `unbox` requiring that Γ' be token-free. However, we can easily write a program that is equivalent to *leakySums2*, that is well-typed in Simply RaTT using tupling, which corresponds to defining *leakySums2* mutually recursively with *map*. Moreover, this side condition for `unbox` was dropped in Lively RaTT as it is incompatible with the extension by temporal inductive types.

Finally, there is an interesting trade-off in all four calculi in terms of their syntactic properties such as eta-expansion and local soundness/completeness. The potential lack of these syntactic properties has no bearing on the semantic soundness results for these calculi, but it may be counterintuitive to a programmer using the language.

For example, typing in Simply RaTT is closed under certain eta-expansions involving \square , which are no longer well-typed in λ_{\checkmark} because of the typing rule for `unbox`. For example, we have

$$f : A \rightarrow \square B \vdash \lambda x. \text{box}(\text{unbox}(f x)) : A \rightarrow \square B$$

in Simply RaTT but not in λ_{\checkmark} . However, λ_{\checkmark} has a slightly different eta-expansion for this type instead:

$$f : A \rightarrow \square B \vdash_{\checkmark} \lambda x. \text{let } x' = \text{unbox}(f x) \text{ in } \text{box } x' : A \rightarrow \square B$$

which matches the eta-expansion in Krishnaswami's calculus:

$$f : A \rightarrow \square B \text{ now } \vdash \lambda x. \text{let } \text{stable}(x') = f x \text{ in } \text{stable } x' : A \rightarrow \square B$$

On the other hand, because λ_{\checkmark} lifts Simply RaTT's restrictions on tokens, λ_{\checkmark} is closed under several types of eta-expansions that are not well-typed in Simply RaTT. For example,

$$\begin{aligned} x : \square(\square A) \vdash_{\checkmark} \text{box}(\text{box}(\text{unbox}(\text{unbox } x))) &: \square(\square A) \\ x : \bigcirc(\bigcirc A) \vdash_{\checkmark} \text{delay}(\text{delay}(\text{adv}(\text{adv } x))) &: \bigcirc(\bigcirc A) \\ f : \bigcirc(A \rightarrow B) \vdash_{\checkmark} \text{delay}(\lambda x. \text{adv } f x) &: \bigcirc(A \rightarrow B) \end{aligned}$$

In return, both Krishnaswami’s calculus and λ_{\llcorner} lack local soundness and completeness for the \square type. For instance, from $\Gamma \vdash_{\llcorner} t : \square A$ we can obtain $\Gamma \vdash_{\llcorner} \text{unbox } t : A$ in λ_{\llcorner} , but from $\Gamma \vdash_{\llcorner} t : A$, we cannot construct a term $\Gamma \vdash_{\llcorner} t' : \square A$. By contrast, the use of the \sharp token ensures that Simply RaTT enjoys these local soundness and completeness properties. However, Simply RaTT can only allow one such token and must thus trade off eta-expansion as show above in order to avoid space leaks (cf. the example term above).

In summary, we argue that our typing system and syntax are simpler than both the work of Krishnaswami (2013) and Bahr *et al.* (2019, 2021). These simplifications are meant to make the language easier to use and integrate more easily with mainstream languages like Haskell, while still maintaining the strong memory guarantees of the earlier calculi.

5 Metatheory

In the previous section, we have presented Rattus’s core calculus and stated its three central operational properties in Theorems 4.2 and 4.3: productivity, causality, and absence of implicit space leaks. Note that the absence of space leaks follows from these theorems because the operational semantics already ensures this memory property by means of garbage collecting the ‘now’ heap after each step. Since the proof of Theorems 4.2 and 4.3 is fully formalised in the accompanying Coq proofs, we only give a high-level overview of the underlying proof technique here.

We prove the above-mentioned theorems by establishing a semantic soundness property. For productivity, our soundness property must imply that the evaluation semantics $\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$ converges for each well-typed term t , and for causality, the soundness property must imply that this is also the case if t contains references to heap locations in σ .

To obtain such a soundness result, we construct a *Kripke logical relation* that incorporates these properties. Generally speaking, a Kripke logical relation constructs for each type A a relation $\llbracket A \rrbracket_w$ indexed over some world w with some closure conditions when the index w changes. In our case, $\llbracket A \rrbracket_w$ is a set of terms. Moreover, the index w consists of three components: a number ν to act as a step index (Appel and McAllester, 2001), a store σ to establish the safety of garbage collection, and an infinite sequence $\bar{\eta}$ of future heaps in order to capture the causality property.

A crucial ingredient of a Kripke logical relation is the ordering on the indices. The ordering on the number ν is the standard ordering on numbers. For heaps, we use the standard ordering on partial maps: $\eta \sqsubseteq \eta'$ iff $\eta(l) = \eta'(l)$ for all $l \in \text{dom}(\eta)$. Infinite sequences of heaps are ordered pointwise according to \sqsubseteq . Moreover, we extend the ordering to stores in two different ways:

$$\frac{\eta_N \sqsubseteq \eta'_N \quad \eta_L \sqsubseteq \eta'_L}{\eta_N \checkmark \eta_L \sqsubseteq \eta'_N \checkmark \eta'_L} \qquad \frac{\sigma \sqsubseteq \sigma'}{\sigma \sqsubseteq_{\checkmark} \sigma'} \qquad \frac{\eta \sqsubseteq \eta'}{\eta \sqsubseteq_{\checkmark} \eta'' \checkmark \eta'}$$

That is, \sqsubseteq is the pointwise extension of the order on heaps to stores, whereas \sqsubseteq_{\checkmark} is more general and permits introducing an arbitrary ‘now’ heap if none is present.

Given these orderings, we define two logical relations, the value relation $\mathcal{V}_{\nu} \llbracket A \rrbracket_{\sigma}^{\bar{\eta}}$ and the term relation $\mathcal{T}_{\nu} \llbracket A \rrbracket_{\sigma}^{\bar{\eta}}$. Both are defined in Figure 9 by well-founded recursion according to

$$\begin{aligned}
\mathcal{V}_v[\text{Int}]_{\bar{\eta}}^{\bar{\eta}} &= \{\bar{n} \mid n \in \mathbb{Z}\}, \\
\mathcal{V}_v[1]_{\bar{\eta}}^{\bar{\eta}} &= \{\langle \rangle\}, \\
\mathcal{V}_v[A \times B]_{\bar{\eta}}^{\bar{\eta}} &= \{\langle v_1, v_2 \rangle \mid v_1 \in \mathcal{V}_v[A]_{\bar{\eta}}^{\bar{\eta}} \wedge v_2 \in \mathcal{V}_v[B]_{\bar{\eta}}^{\bar{\eta}}\}, \\
\mathcal{V}_v[A + B]_{\bar{\eta}}^{\bar{\eta}} &= \{\text{in}_1 v \mid v \in \mathcal{V}_v[A]_{\bar{\eta}}^{\bar{\eta}}\} \cup \{\text{in}_2 v \mid v \in \mathcal{V}_v[B]_{\bar{\eta}}^{\bar{\eta}}\}, \\
\mathcal{V}_v[A \rightarrow B]_{\bar{\eta}}^{\bar{\eta}} &= \left\{ \lambda x.t \mid \forall v' \leq v, \sigma' \sqsupseteq \text{gc}(\sigma), \bar{\eta}' \sqsupseteq \bar{\eta}. \forall u \in \mathcal{V}_{v'}[A]_{\bar{\eta}'}^{\bar{\eta}'}. t[u/x] \in \mathcal{V}_{v'}[B]_{\bar{\eta}'}^{\bar{\eta}'} \right\}, \\
\mathcal{V}_v[\Box A]_{\bar{\eta}}^{\bar{\eta}} &= \{\text{box } t \mid \forall \bar{\eta}'. t \in \mathcal{V}_v[A]_{\bar{\eta}'}^{\bar{\eta}'}\}, \\
\mathcal{V}_0[\bigcirc A]_{\bar{\eta}}^{\bar{\eta}} &= \{l \mid l \in \text{Loc}\} \\
\mathcal{V}_{v+1}[\bigcirc A]_{\bar{\eta}}^{\bar{\eta}} &= \{l \mid \sigma(l) \in \mathcal{V}_v[A]_{\text{gc}(\sigma)\checkmark\eta}^{\bar{\eta}}\}, \\
\mathcal{V}_v[\text{Fix } \alpha.A]_{\bar{\eta}}^{\bar{\eta}} &= \left\{ \text{into}(v) \mid v \in \mathcal{V}_v[A[\bigcirc(\text{Fix } \alpha.A)/\alpha]]_{\bar{\eta}}^{\bar{\eta}} \right\} \\
\mathcal{V}_v[A]_{\bar{\eta}}^{\bar{\eta}} &= \left\{ t \mid \forall \sigma' \sqsupseteq \checkmark \sigma. \exists \sigma'', v. \langle t; \sigma' \rangle \Downarrow \langle v; \sigma'' \rangle \wedge v \in \mathcal{V}_v[A]_{\sigma''}^{\bar{\eta}} \right\} \\
\mathcal{E}_v[\cdot]_{\bar{\eta}}^{\bar{\eta}} &= \{\star\} && \text{GARBAGE COLLECTION:} \\
\mathcal{E}_v[\Gamma, x:A]_{\bar{\eta}}^{\bar{\eta}} &= \left\{ \gamma[x \mapsto v] \mid \gamma \in \mathcal{E}_v[\Gamma]_{\bar{\eta}}^{\bar{\eta}}, v \in \mathcal{V}_v[A]_{\bar{\eta}}^{\bar{\eta}} \right\} && \text{gc}(\eta_L) = \eta_L \\
\mathcal{E}_v[\Gamma, \checkmark]_{\eta_N\checkmark\eta_L}^{\bar{\eta}} &= \mathcal{E}_{v+1}[\Gamma]_{\eta_N}^{\bar{\eta}} && \text{gc}(\eta_N\checkmark\eta_L) = \eta_L
\end{aligned}$$

Fig. 9. Logical relation.

the lexicographic ordering on the triple $(v, |A|, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation:

$$\begin{aligned}
|\alpha| &= |\bigcirc A| = |\text{Int}| = |1| = 1 \\
|A \times B| &= |A + B| = |A \rightarrow B| = 1 + |A| + |B| \\
|\Box A| &= |\text{Fix } \alpha.A| = 1 + |A|
\end{aligned}$$

In the definition of the logical relation, we use the notation $\eta; \bar{\eta}$ to denote an infinite sequence of heaps that starts with the heap η and then continues as the sequence $\bar{\eta}$. Moreover, we use the notation $\sigma(l)$ to denote $\eta_L(l)$ if σ is of the form η_L or $\eta_N\checkmark\eta_L$.

The crucial part of the logical relation that ensures both causality and the absence of space leaks is the case for $\bigcirc A$. The value relation of $\bigcirc A$ at store index σ is defined as all heap locations that map to computations in the term relation of A but at the store index $\text{gc}(\sigma)\checkmark\eta$. Here, $\text{gc}(\sigma)$ denotes the garbage collection of the store σ as defined in Figure 9. It simply drops the ‘now’ heap if present. To see how this definition captures causality, we have to look at the index $\eta; \bar{\eta}$ of future heaps. It changes to the index $\bar{\eta}$, that is, all future heaps are one time step closer, and the very first future heap η becomes the new ‘later’ heap in the store index $\text{gc}(\sigma)\checkmark\eta$, whereas the old ‘later’ heap in σ becomes the new ‘now’ heap.

The central theorem that establishes type soundness is the so-called *fundamental property* of the logical relation. It states that well-typed terms are in the term relation. For the induction proof of this property, we also need to consider open terms and to this end, we also need a corresponding context relation $\mathcal{E}_v[\Gamma]_{\bar{\eta}}^{\bar{\eta}}$, which is given in Figure 9.

Theorem 5.1 (Fundamental Property). *If $\Gamma \vdash_v t : A$, and $\gamma \in \mathcal{E}_v[\Gamma]_{\bar{\eta}}^{\bar{\eta}}$, then $t\gamma \in \mathcal{V}_v[A]_{\bar{\eta}}^{\bar{\eta}}$.*

data $\bigcirc a = \text{Delay } a$	data $\square a = \text{Box } a$	class <i>StableInternal</i> a where
<code>delay</code> $:: a \rightarrow \bigcirc a$	<code>box</code> $:: a \rightarrow \square a$	class <i>StableInternal</i> $a \Rightarrow \text{Stable } a$ where
<code>delay</code> $x = \text{Delay } x$	<code>box</code> $x = \text{Box } x$	
<code>adv</code> $:: \bigcirc a \rightarrow a$	<code>unbox</code> $:: \square a \rightarrow a$	
<code>adv</code> $(\text{Delay } x) = x$	<code>unbox</code> $(\text{Box } d) = d$	

Fig. 10. Implementation of Rattus primitives.

The proof of the fundamental property is a lengthy but entirely standard induction on the typing relation $\Gamma \vdash_{\checkmark} t : A$. Both Theorems 4.2 and 4.3 are then proved using the above theorem.

6 Embedding Rattus in Haskell

Our goal with Rattus is to combine the operational guarantees provided by modal FRP with the practical benefits of FRP libraries. Because of its Fitch-style typing rules, we cannot implement Rattus as just a library of combinators. Instead, we rely on a combination of a very simple library that implements the primitives of the language together with a compiler plugin that performs additional checks. In addition, we also have to implement the operational semantics of Rattus, which is by default call-by-value and thus different from Haskell’s. This discrepancy in the operational semantics suggest a deep embedding of the language. However, in order to minimise syntactic overhead and to seamlessly integrate Rattus with its host language, we chose a shallow embedding and instead rely on the compiler plugin to perform the necessary transformations to ensure the correct operational behaviour of Rattus programs.

We start with a description of the implementation followed by an illustration of how the implementation is used in practice.

6.1 Implementation of Rattus

At its core, our implementation consists of a very simple library that implements the primitives of Rattus (`delay`, `adv`, `box` and `unbox`) so that they can be readily used in Haskell code. The library is given in its entirety in Figure 10. Both \bigcirc and \square are simple wrapper types, each with their own wrap and unwrap function. The constructors *Delay* and *Box* are not exported by the library, that is, \bigcirc and \square are treated as abstract types.

If we were to use these primitives as provided by the library, we would end up with the problems illustrated in Section 2. Such an implementation of Rattus would enjoy none of the operational properties we have proved. To make sure that programs use these primitives according to the typing rules of Rattus, our implementation has a second component: a plugin for the GHC Haskell compiler that enforces the typing rules of Rattus.

The design of this plugin follows the simple observation that any Rattus program is also a Haskell program but with more restrictive rules for variable scope and for where Rattus’s primitives may be used. So type-checking a Rattus program boils down to first type-checking it as a Haskell program and then checking that it follows the stricter variable scope rules. That means, we must keep track of when variables fall out of scope due to the

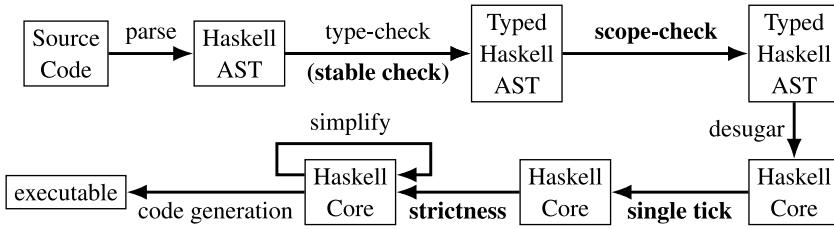


Fig. 11. Compiler phases of GHC (simplified) extended with Rattus plugin (highlighted in bold).

use of `delay`, `adv` and `box`, but also due to guarded recursion. Additionally, we must make sure that both guarded recursive calls and `adv` only appear in a context where \checkmark is present.

To enforce these additional simple scope rules, we make use of GHC’s plugin API which allows us to customise part of GHC’s compilation pipeline. The different phases of GHC are illustrated in Figure 11 with the additional passes performed by the Rattus plugin highlighted in bold.

After type-checking the Haskell abstract syntax tree (AST), GHC passes the resulting typed AST on to the scope-checking component of the Rattus plugin, which checks the above-mentioned stricter scoping rules. GHC then desugars the typed AST into the intermediate language *Core*. GHC then performs a number of transformation passes on this intermediate representation, and the first two of these are provided by the Rattus plugin: first, we exhaustively apply the two rewrite rules from Section 4.2.1 to transform the program into a single-tick form according to the typing rules of λ_{\checkmark} . Then we transform the resulting code so that Rattus programs adhere to the call-by-value semantics. To this end, the plugin’s *strictness* pass transforms all lambda abstractions so that they evaluate their arguments to weak head normal form, and all let bindings so that they evaluate the bound expression into weak head normal form. This is achieved by transforming lambda abstractions and let bindings as follows:

$$\lambda x.t \text{ is replaced by } \lambda x.\mathbf{case} \ x \ \mathbf{of} \ _ \rightarrow t, \text{ and}$$

$$\mathbf{let} \ x = s \ \mathbf{in} \ t \text{ is replaced by } \mathbf{case} \ s \ \mathbf{of} \ x \rightarrow t$$

In Haskell Core, case expressions always evaluate the scrutinee to weak head normal form even if there is only a default clause. Hence, this transformation will force the evaluation of x in the lambda abstraction $\lambda x.t$, and the evaluation of s in the let binding $\mathbf{let} \ x = s \ \mathbf{in} \ t$. In addition, this *strictness* pass also checks that Rattus code only uses strict data types and issues a warning if lazy data types are used, for example, Haskell’s standard list and pair types. Taken together, the transformations and checks performed by the *strictness* pass ensure that lambda abstractions, let bindings and data constructors follow the operational semantics of λ_{\checkmark} . The remaining components of the language are either implemented directly as Haskell functions (`delay`, `adv`, `box` and `unbox`) and thus require no transformation or use Haskell’s recursion syntax that matches the semantics of the corresponding fixed point combinator in λ_{\checkmark} .

However, the Haskell implementation of `delay` and `adv` do not match the operational semantics of λ_{\checkmark} exactly. Instead of explicit allocation of delayed computations on a heap that then enables the eager garbage collection strategy of the λ_{\checkmark} operational semantics,

```

{-# OPTIONS -fplugin=Rattus.Plugin #-}
import Rattus
import Rattus.Stream
import Rattus.ToHaskell
{-# ANN sums Rattus #-}
sums :: Str Int → Str Int
sums = scan (box (+)) 0

main = loop (runTransducer sums)
where loop (Trans t) = do
  input ← readLn
  let (result, next) = t input
  print result
  loop next

```

Fig. 12. Complete Rattus program.

we rely on Haskell’s lazy evaluation for delay and GHC’s standard garbage collection implementation. Since our metatheoretical results show that old temporal data is no longer referenced after each time step, such data will indeed be garbage collected by the GHC runtime system.

Not pictured in Figure 11 is a second scope-checking pass that is performed after the *strictness* pass. After the *single tick* pass and thus also after the *strictness* pass, we expect that the code is typable according to the more restrictive typing rules of λ_{\checkmark} . This second scope-checking pass checks this invariant for the purpose of catching implementation bugs in the Rattus plugin. The Core intermediate language is *much* simpler than the full Haskell language, so this second scope-checking pass is much easier to implement and much less likely to contain bugs. In principle, we could have saved ourselves the trouble of implementing the much more complicated scope-checking at the level of the typed Haskell AST. However, by checking at this earlier stage of the compilation pipeline, we can provide much more helpful type error messages.

One important component of checking variable scope is checking whether types are stable. This is a simple syntactic check: a type τ is stable if all occurrences of \bigcirc or function types in τ are nested under a \square . However, Rattus also supports polymorphic types with type constraints such as in the *const* function:

```

const :: Stable a ⇒ a → Str a
const x = x :: delay (const x)

```

The *Stable* type class is defined as a primitive in the Rattus library (see Figure 10). The library does not export the underlying *StableInternal* type class so that the user cannot declare any instances for *Stable*. Our library does not declare instances of the *Stable* class either. Instead, such instances are derived by the Rattus plugin that uses GHC’s type checker plugin API, which allows us to provide limited customisation to GHC’s type-checking phase (see Figure 11). Using this API, one can give GHC a custom procedure for resolving type constraints. Whenever GHC’s type checker finds a constraint of the form *Stable* τ , it will send it to the Rattus plugin, which will resolve it by performing the above-mentioned syntactic check on τ .

6.2 Using Rattus

To write Rattus code inside Haskell, one must use GHC with the flag `-fplugin=Rattus.Plugin`, which enables the Rattus plugin described above. Figure 12

shows a complete program that illustrates the interaction between Haskell and Rattus. The language is imported via the *Rattus* module, with the *Rattus.Stream* module providing a stream library (of which we have seen an excerpt in Figure 1). The program contains only one Rattus function, *summing*, which is indicated by an annotation. This function uses the *scan* combinator to define a stream transducer that sums up its input stream. Finally, we use the *runTransducer* function that is provided by the *Rattus.ToHaskell* module. It turns a stream function of type $Str\ a \rightarrow Str\ b$ into a Haskell value of type $Trans\ a\ b$ defined as follows:

```
data  $Trans\ a\ b = Trans\ (a \rightarrow (b, Trans\ a\ b))$ 
```

This allows us to run the stream function step by step as illustrated in the main function: It reads an integer from the console passes it on to the stream function, prints out the response and then repeats the process.

Alternatively, if a module contains only Rattus definitions, we can use the annotation

```
{-# ANN module Rattus #-}
```

to declare that all definitions in a module are to be interpreted as Rattus code.

7 Related work

Embedded FRP languages. The central ideas of FRP were originally developed for the language Fran (Elliott and Hudak, 1997) for reactive animation. These ideas have since been developed into general-purpose libraries for reactive programming, most prominently the Yampa library (Nilsson *et al.*, 2002) for Haskell, which has been used in a variety of applications including games, robotics, vision, GUIs and sound synthesis.

More recently Ploeg and Claessen (2015) have developed the *FRPNow!* library for Haskell, which – like Fran – uses behaviours and events as FRP primitives (as opposed to signal functions) but carefully restricts the API to guarantee causality and the absence of implicit space leaks. To argue for the latter, the authors construct a denotational model and show using a logical relation that their combinators are not ‘inherently leaky’. The latter does not imply the absence of space leaks, but rather that in principle it can be implemented without space leaks.

While these FRP libraries do not allow direct manipulation of signals since they lack a bespoke type system to make that safe, they do have a practical advantage: their reliance on the host language’s type system makes their implementation and maintenance markedly easier. Moreover, by embedding FRP libraries in host languages with a richer type system, such as full dependent types, one can still obtain some operational guarantees, including productivity (Sculthorpe and Nilsson, 2009).

Modal FRP calculi. The idea of using modal type operators for reactive programming goes back to Jeffrey (2012), Krishnaswami and Benton (2011), and Jeltsch (2013). One of the inspirations for Jeffrey (2012) was to use linear temporal logic (Pnueli, 1977) as a programming language through the Curry–Howard isomorphism. The work of Jeffrey and Jeltsch has mostly been based on denotational semantics, and to our knowledge Krishnaswami and Benton (2011), Krishnaswami *et al.* (2012), Krishnaswami (2013), Cave *et al.* (2014), and

Bahr *et al.* (2019, 2021) are the only works giving operational guarantees. In addition, the calculi of Cave *et al.* (2014) and Bahr *et al.* (2021) can encode liveness properties in types by distinguishing between data (least fixed points, e.g., events that must happen within a finite number of steps) and codata (greatest fixed points, e.g., streams). With the help of both greatest and least fixed points, one can express liveness properties of programs in their types (e.g., a server that will always respond within a finite number of time steps). Temporal logic has also been used directly as a specification language for property-based testing and runtime verification of FRP programs (Perez and Nilsson, 2020).

Guarded recursive types and the guarded fixed point combinator originate with Nakano (2000) but have since been used for constructing logics for reasoning about advanced programming languages (Birkedal *et al.*, 2011) using an abstract form of step-indexing (Appel and McAllester, 2001). The Fitch-style approach to modal types (Fitch, 1952; Clouston, 2018) has been used for guarded recursion in Clocked Type Theory (Bahr *et al.*, 2017), where contexts can contain multiple, named ticks. Ticks can be used for reasoning about guarded recursive programs. The denotational semantics of Clocked Type Theory (Mannaa and Møgelberg, 2018) reveals the difference from the more standard dual-context approaches to modal logics, such as Dual Intuitionistic Linear Logic (Barber, 1996): in the latter, the modal operator is implicitly applied to the type of all variables in one context, in the Fitch-style, placing a tick in a context corresponds to applying a *left adjoint* to the modal operator to the context. Guatto (2018) introduced the notion of time warp and the warping modality, generalising the delay modality in guarded recursion, to allow for a more direct style of programming for programs with complex input–output dependencies.

Space leaks. The work by Krishnaswami (2013) and Bahr *et al.* (2019, 2021) is the closest to the present work. Both present a modal FRP language with a garbage collection result similar to ours. Krishnaswami (2013) pioneered this approach to prove the absence of implicit space leaks but also implemented a compiler for his language, which translates FRP programs into JavaScript. Bahr *et al.* (2019) were the first to mechanise the metatheory for a modal FRP calculus, and our mechanisation is based on their work. For a more detailed comparison with these calculi, see Section 4.5.

Krishnaswami *et al.* (2012) approached the problem of space leaks with an affine type system that keeps track of permission tokens for allocating a stream cons cell. This typing discipline ensures that space usage is bounded by the number of provided permission tokens and thus provides more granular static checks of space usage.

Synchronous dataflow languages, such Esterel (Berry and Cosserat, 1985), Lustre (Caspi *et al.*, 1987) and Lucid Synchronic (Pouzet, 2006), provide even stronger static guarantees – not only on space usage but also on time usage. This feature has made these languages attractive in resource-constrained environments such as hardware synthesis and embedded control software. Their computational model is based on a fixed network of stream processing nodes where each node consumes and produces a statically known number of primitive values at each discrete time step. As a trade-off for these static guarantees, synchronous dataflow languages support neither time-varying values of arbitrary types nor dynamic switching.

8 Discussion and future work

We have shown that modal FRP with strong operational guarantees can be seamlessly integrated into the Haskell programming language. Two main ingredients are central to achieving this integration: (1) the use of Fitch-style typing to simplify the syntax for interacting with the two modalities and (2) lifting some of the restrictions found in previous work on Fitch-style typing systems.

This paper opens up several avenues for future work both on the implementation side and the underlying theory. We chose Haskell as our host language as it has a compiler extension API that makes it easy for us to implement Rattus and convenient for programmers to start using Rattus with little friction. However, we think that implementing Rattus in call-by-value languages like OCaml or F# should be easily achieved by a simple post-processing step that checks the Fitch-style variable scope. This can be done by an external tool (not unlike a linter) that does not need to be integrated into the compiler. Moreover, while the use of the type class *Stable* is convenient, it is not necessary as we can always use the \square modality instead (cf. *const* versus *constBox*). When a program transformation approach is not feasible or not desirable, one can also use λ_{\surd} rather than λ_{\swarrow} as the underlying calculus. We suspect that most function definitions do not need the flexibility of λ_{\swarrow} and those that do can be transformed by the programmer with only little syntactic clutter. One could imagine that the type checker could suggest these transformations to the programmer rather than silently performing them itself.

FRP is not the only possible application of Fitch-style type systems. However, most of the interest in Fitch-style system has been in logics and dependent type theory (Clouston, 2018; Birkedal *et al.*, 2018; Bahr *et al.*, 2017; Borghuis, 1994) as opposed to programming languages. Rattus is to our knowledge the first implementation of a Fitch-style programming language. We would expect that programming languages for information control flow (Kavvos, 2019) and recent work on modalities for pure computations Chaudhury and Krishnaswami (2020) admit a Fitch-style presentation and could be implemented similarly to Rattus.

While Fitch-style modal FRP languages promise strong static guarantees with only moderate syntactic overhead, we still lack empirical evidence that they help practical adoption of FRP. Our goal with Rattus is to provide an implementation with a low entry barrier and with access to a mature and practical software development ecosystem. We hope that this enables experimentation with Rattus not only by researchers, but also students and practitioners.

For example, we have looked at only a small fragment of Yampa's comprehensive arrowised FRP library. A thorough re-implementation of Yampa in Rattus could provide a systematic comparison of their relative expressiveness. For such an effort, we expect that the signal function type be refined to include the \square modality:

data $SF\ a\ b = SF\ !(\square(Float \rightarrow a \rightarrow (\bigcirc(SF\ a\ b) \otimes b)))$

This enables the implementation of Yampa's powerful switch combinators in Rattus, which among other things can be used to stop a signal function and then resume it (with all its internal state) at a later point. By contrast, it is not clear how such a switch combinator can be provided in an FRP library with first-class streams as presented in Section 3.1.

Part of the success of FRP libraries such as Yampa and FRPNow! is due to the fact that they provide a rich and highly optimised API that integrates well with its host language. In this paper, we have shown that Rattus can be seamlessly embedded in Haskell, but more work is required to design a comprehensive library and to perform the low-level optimisations that are often necessary to obtain good real-world performance. For example, our definition of signal functions in Section 3.3 resembles the semantics of Yampa's signal functions, but Yampa's signal functions are implemented as generalised algebraic data types (GADTs) that can handle some special cases much more efficiently and enable dynamic optimisations.

In order to devise sound optimisations for Rattus, we also require suitable coinductive reasoning principles. For example, we might want to make use of Haskell's rewrite rules to perform optimisations such as map fusion as expressed in the following equation:

$$\text{map } f (\text{map } g \text{ } xs) = \text{map } (\text{box } (\text{unbox } f \circ \text{unbox } g)) \text{ } xs$$

Such an equation could be proved sound using step-indexed logical relations, not unlike the one we presented in Section 5. For this to scale, however, we need more high-level reasoning principles such as a sound coinductive axiomatisation of bisimilarity or more generally a suitable type theory with guarded recursion (Møgelberg and Veltri, 2019).

Conflicts of interest

None.

Supplementary material

For supplementary material for this article, please visit <https://doi.org/10.1017/S0956796822000132>.

References

- Abramsky, S., Honda, K. & McCusker, G. (1998) A fully abstract game semantics for general references. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 98CB36226)*. Indianapolis, IN, USA: IEEE Computer Society, pp. 334–344.
- 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., Grathwohl, H. B. & Møgelberg, R. E. (2017) The clocks are ticking: No more delays! In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*, Reykjavik, Iceland, June 20–23, 2017. Washington, DC, USA: IEEE Computer Society, pp. 1–12.
- Bahr, P., Graulund, C. U. & Møgelberg, R. E. (2019) Simply RaTT: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.* **3**(ICFP), 1–27.
- Bahr, P., Graulund, C. U. & Møgelberg, R. E. (2021) Diamonds are not forever: Liveness in reactive programming with guarded recursion. In *POPL 2021*, to appear.
- Barber, A. (1996) Dual intuitionistic linear logic. Technical report. University of Edinburgh. Edinburgh, UK.
- Berry, G. & Cosserat, L. (1985) The estereel synchronous programming language and its mathematical semantics. In *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. (2018) Modal dependent type theory and dependent right adjoints. *arXiv:1804.05236 [cs]*. 00000 arXiv: 1804.05236.
- Birkedal, L., Møgelberg, R. E., Schwinghammer, J. & Støvring, K. (2011) First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *Proceedings of LICS*. Washington, DC, USA: IEEE Computer Society, pp. 55–64.
- Borghuis, V. A. J. (1994) *Coming to Terms with Modal Logic: On the Interpretation of Modalities in Typed Lambda-Calculus*. PhD Thesis. Technische Universiteit Eindhoven, 00034.
- Caspi, P., Pilaud, D., Halbwachs, N. & Plaice, J. A. (1987) Lustre: A declarative language for real-time programming. In *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. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA: ACM, pp. 361–372.
- Chaudhury, V. & Krishnaswami, N. (2020) Recovering purity with comonads and capabilities. In *ICFP 2020*, to appear.
- Clouston, R. (2018) Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*. Cham: Springer. Springer International Publishing, pp. 258–275.
- Elliott, C. & Hudak, P. (1997) Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, pp. 263–273.
- Fitch, F. B. (1952) *Symbolic Logic, An Introduction*. New York, NY, USA: Ronald.
- Guatto, A. (2018) A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, pp. 482–491.
- Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2004) Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*. Springer Berlin/Heidelberg.
- Jeffrey, A. (2012) LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012*, Philadelphia, PA, USA, January 24, 2012. Philadelphia, PA, USA: ACM, pp. 49–60.
- Jeffrey, A. (2014) Functional reactive types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. New York, NY, USA: ACM, pp. 54:1–54:9.
- Jeltsch, W. (2013) Temporal logic with “until”, functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*. New York, NY, USA: ACM, pp. 69–78.
- Järvi, J., Marcus, M., Parent, S., Freeman, J. & Smith, J. N. (2008) Property Models: From Incidental Algorithms to Reusable Components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, pp. 89–98.
- Kavvos, G. A. (2019) Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.* 3(POPL), 20:1–20:29, 00000.
- Krishnaswami, N. R. (2013) Higher-order functional reactive programming without spacetime leaks. In *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. In *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. In *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.

- Mannaa, B. & Møgelberg, R. E. (2018) The clocks they are adjunctions: Denotational semantics for clocked type theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018*, July 9–12, 2018, Oxford, UK. New York, NY, USA, pp. 23:1–23:17.
- Møgelberg, R. E. & Veltri, N. (2019) Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.* **3**(POPL), 4:1–4:29.
- Nakano, H. (2000) A modality for recursion. In *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. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. New York, NY, USA: ACM, pp. 51–64.
- Parent, S. (2006) A possible future for software development. Keynote talk at the Workshop of Library-Centric Software Design at OOPSLA’06.
- Paterson, R. (2001) A new notation for arrows. *ACM SIGPLAN Not.* **36**(10), 229–240, 00234.
- Perez, I., Bärenz, M. & Nilsson, H. (2016) Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell*. New York, NY, USA: Association for Computing Machinery, pp. 33–44.
- Perez, I. & Nilsson, H. (2020) Runtime verification and validation of functional reactive systems. *J. Funct. Program.* **30**.
- van der Ploeg, A. & Claessen, K. (2015) Practical principled FRP: forget the past, change the future, FRPNow! In *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. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, pp. 46–57.
- Pouzet, M. (2006) Lucid synchrone, version 3. *Tutorial and Reference Manual. Université Paris-Sud, LRI*. **1**, 25.
- Sculthorpe, N. & Nilsson, H. (2009) Safe functional reactive programming through dependent types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, pp. 23–34.

A Multi-tick calculus

In this appendix, we give the proof of Theorem 4.1, that is, we show that the program transformation described in Section 4.2.1 indeed transforms any closed λ_{\checkmark} term into a closed λ_{\checkmark} term.

Figure 13 gives the context formation rules of λ_{\checkmark} ; the only difference compared to λ_{\checkmark} is the rule for adding ticks, which has a side condition so that there may be no more than one tick. Figure 14 lists the full set of typing rules of λ_{\checkmark} . Compared to λ_{\checkmark} (cf. Figure 5), the only rules that have changed are the rule for lambda abstraction, and the rule for delay. Both rules transform the context Γ to $|\Gamma|$, which removes the \checkmark in Γ if it has one:

$$|\Gamma| = \Gamma \quad \text{if } \Gamma \text{ is tick-free}$$

$$|\Gamma, \checkmark, \Gamma'| = \Gamma^{\square}, \Gamma'$$

We define the rewrite relation \longrightarrow as the least relation that is closed under congruence and the following rules:

$$\frac{}{\emptyset \vdash_{\checkmark}} \quad \frac{\Gamma \vdash_{\checkmark}}{\Gamma, x : A \vdash_{\checkmark}} \quad \frac{\Gamma \vdash_{\checkmark} \quad \Gamma \text{ tick-free}}{\Gamma, \checkmark \vdash_{\checkmark}}$$

Fig. 13. Well-formed contexts of λ_{\checkmark} .

$$\frac{\Gamma, x : A, \Gamma' \vdash_{\checkmark} \quad \Gamma' \text{ tick-free or } A \text{ stable}}{\Gamma, x : A, \Gamma' \vdash_{\checkmark} x : A} \quad \frac{\Gamma \vdash_{\checkmark}}{\Gamma \vdash_{\checkmark} \langle \rangle : 1} \quad \frac{n \in \mathbb{Z}}{\Gamma \vdash_{\checkmark} \bar{n} : \text{Int}}$$

$$\frac{\Gamma \vdash_{\checkmark} s : \text{Int} \quad \Gamma \vdash_{\checkmark} t : \text{Int}}{\Gamma \vdash_{\checkmark} s + t : \text{Int}} \quad \frac{|\Gamma|, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \lambda x.t : A \rightarrow B} \quad \frac{\Gamma \vdash_{\checkmark} s : A \quad \Gamma, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \text{let } x = s \text{ in } t : B}$$

$$\frac{\Gamma \vdash_{\checkmark} t : A \rightarrow B \quad \Gamma \vdash_{\checkmark} t' : A}{\Gamma \vdash_{\checkmark} t t' : B} \quad \frac{\Gamma \vdash_{\checkmark} t : A \quad \Gamma \vdash_{\checkmark} t' : B}{\Gamma \vdash_{\checkmark} \langle t, t' \rangle : A \times B}$$

$$\frac{\Gamma \vdash_{\checkmark} t : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash_{\checkmark} \pi_i t : A_i} \quad \frac{\Gamma \vdash_{\checkmark} t : A_i \quad i \in \{1, 2\}}{\Gamma \vdash_{\checkmark} \text{in}_i t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash_{\checkmark} t_i : B \quad \Gamma \vdash_{\checkmark} t : A_1 + A_2 \quad i \in \{1, 2\}}{\Gamma \vdash_{\checkmark} \text{case } t \text{ of } \text{in}_1 x.t_1; \text{in}_2 x.t_2 : B} \quad \frac{|\Gamma|, \checkmark \vdash_{\checkmark} t : A}{\Gamma \vdash_{\checkmark} \text{delay } t : \bigcirc A}$$

$$\frac{\Gamma \vdash_{\checkmark} t : \bigcirc A \quad \Gamma' \text{ tick-free}}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{adv } t : A} \quad \frac{\Gamma \vdash_{\checkmark} t : \square A}{\Gamma \vdash_{\checkmark} \text{unbox } t : A} \quad \frac{\Gamma^{\square} \vdash_{\checkmark} t : A}{\Gamma \vdash_{\checkmark} \text{box } t : \square A}$$

$$\frac{\Gamma \vdash_{\checkmark} t : A[\bigcirc(\text{Fix } \alpha.A)/\alpha]}{\Gamma \vdash_{\checkmark} \text{into } t : \text{Fix } \alpha.A} \quad \frac{\Gamma \vdash_{\checkmark} t : \text{Fix } \alpha.A}{\Gamma \vdash_{\checkmark} \text{out } t : A[\bigcirc(\text{Fix } \alpha.A)/\alpha]} \quad \frac{\Gamma^{\square}, x : \square(\bigcirc A) \vdash_{\checkmark} t : A}{\Gamma \vdash_{\checkmark} \text{fix } x.t : A}$$

Fig. 14. Typing rules of λ_{\checkmark} .

$$\text{delay}(K[\text{adv } t]) \longrightarrow \text{let } x = t \text{ in } \text{delay}(K[\text{adv } x]) \quad \text{if } t \text{ is not a variable}$$

$$\lambda x.K[\text{adv } t] \longrightarrow \text{let } y = \text{adv } t \text{ in } \lambda x.(K[y])$$

where K is a term with a single occurrence of a hole \square that is not in the scope of delay , adv , box , fix , or a lambda abstraction. Formally, K is generated by the following grammar:

$$K ::= \square \mid K t \mid t K \mid \text{let } x = K \text{ in } t \mid \text{let } x = t \text{ in } K \mid \langle K, t \rangle \mid \langle t, K \rangle \mid \text{in}_1 K \mid \text{in}_2 K \mid \pi_1 K \mid \pi_2 K$$

$$\mid \text{case } K \text{ of } \text{in}_1 x.s; \text{in}_2 x.t \mid \text{case } s \text{ of } \text{in}_1 x.K; \text{in}_2 x.t \mid \text{case } s \text{ of } \text{in}_1 x.t; \text{in}_2 x.K$$

$$\mid K + t \mid t + K \mid \text{into } K \mid \text{out } K \mid \text{unbox } K$$

We write $K[t]$ to substitute the unique hole \square in K with the term t .

In the following, we show that for each $\vdash_{\checkmark} t : A$, if we exhaustively apply the above rewrite rules to t , we obtain a term $\vdash_{\checkmark} t' : A$. We prove this by proving each of the following properties in turn:

- (1) Subject reduction: If $\Gamma \vdash_{\checkmark} s : A$ and $s \longrightarrow t$, then $\Gamma \vdash_{\checkmark} t : A$.
- (2) Exhaustiveness: If t is a normal form for \longrightarrow , then $\vdash_{\checkmark} t : A$ implies $\vdash_{\checkmark} t : A$.
- (3) Strong normalisation: There is no infinite \longrightarrow -reduction sequence.

A.1 Subject reduction

We first show subject reduction (cf. Proposition A.4 below). To this end, we need a number of lemmas:

Lemma A.1 (weakening). *Let $\Gamma_1, \Gamma_2 \vdash_{\mathcal{W}} t : A$ and Γ tick-free. Then $\Gamma_1, \Gamma, \Gamma_2 \vdash_{\mathcal{W}} t : A$.*

Proof By straightforward induction on $\Gamma_1, \Gamma_2 \vdash_{\mathcal{W}} t : A$. ■

Lemma A.2. *Given $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A$ with Γ' tick-free, then there is some type B such that $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } x] : A$.*

Proof We proceed by induction on the structure of K .

- \square : $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{adv } t : A$ and Γ' tick-free implies that $\Gamma \vdash_{\mathcal{W}} t : \bigcirc A$. Moreover, given a fresh variable x , we have that $\Gamma, x : \bigcirc A \vdash_{\mathcal{W}} x : \bigcirc A$, and thus $\Gamma, x : \bigcirc A, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{adv } x : A$ and Γ' .
- $\underline{K} s$: $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] s : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A'$ and $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A' \rightarrow A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } x] : A' \rightarrow A$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A'$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } x] s : A$.
- $\underline{s} K$: $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[s \text{ adv } t] : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A' \rightarrow A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A'$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } x] : A'$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A' \rightarrow A$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} s K[\text{adv } x] : A$.
- let $y = s$ in K : $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{let } y = s \text{ in } K[\text{adv } t] : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A'$ and $\Gamma, \checkmark, \Gamma', y : A' \vdash_{\mathcal{W}} K[\text{adv } t] : A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma', y : A' \vdash_{\mathcal{W}} K[\text{adv } x] : A$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} s : A'$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{let } y = s \text{ in } K[\text{adv } x] : A$.
- let $y = K$ in s : $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{let } y = K[\text{adv } t] \text{ in } s : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma', y : A' \vdash_{\mathcal{W}} s : A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A'$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } x] : A'$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma', y : A' \vdash_{\mathcal{W}} s : A$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathcal{W}} \text{let } y = K[\text{adv } x] \text{ in } s : A$.

The remaining cases follow by induction hypothesis and Lemma A.1 in a manner similar to the cases above. ■

Lemma A.3. *Let $\Gamma, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A$ and Γ' tick-free. Then there is some type B such that $\Gamma \vdash_{\mathcal{W}} \text{adv } t : B$ and $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} K[x] : A$.*

Proof We proceed by induction on the structure of K :

- $\llbracket \cdot \rrbracket$: $\Gamma, \Gamma' \vdash_{\mathcal{W}} \text{adv } t : A$ and Γ' tick-free implies that there must be Γ_1 and Γ_2 such that Γ_2 is tick-free, $\Gamma = \Gamma_1, \checkmark, \Gamma_2$, and $\Gamma_1 \vdash_{\mathcal{W}} t : \bigcirc A$. Hence, $\Gamma_1, \checkmark, \Gamma_2 \vdash_{\mathcal{W}} \text{adv } t : A$. Moreover, $\Gamma, x : A, \Gamma' \vdash_{\mathcal{W}} x : A$ follows immediately by the variable introduction rule.
- $\underline{K}s$: $\Gamma, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t]s : A$ implies that there is some A' with $\Gamma, \Gamma' \vdash_{\mathcal{W}} s : A'$ and $\Gamma, \Gamma' \vdash_{\mathcal{W}} K[\text{adv } t] : A' \rightarrow A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} \text{adv } t : B$ and $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} K[x] : A' \rightarrow A$. Hence, $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} s : A'$, by Lemma A.1, and thus $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} K[x]s : A$.
- $\text{let } y = s \text{ in } K$: $\Gamma, \Gamma' \vdash_{\mathcal{W}} \text{let } y = s \text{ in } K[\text{adv } t] : A$ implies that there is some A' with $\Gamma, \Gamma' \vdash_{\mathcal{W}} s : A'$ and $\Gamma, \Gamma', y : A' \vdash_{\mathcal{W}} K[\text{adv } t] : A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathcal{W}} t : B$ and $\Gamma, x : B, \Gamma', y : A' \vdash_{\mathcal{W}} K[x] : A$. Hence, $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} s : A'$, by Lemma A.1, and thus $\Gamma, x : B, \Gamma' \vdash_{\mathcal{W}} \text{let } y = s \text{ in } K[\text{adv } x] : A$.

The remaining cases follow by induction hypothesis and Lemma A.1 in a manner similar to the cases above. ■

Proposition A.4 (subject reduction). *If $\Gamma \vdash_{\mathcal{W}} s : A$ and $s \rightarrow t$, then $\Gamma \vdash_{\mathcal{W}} t : A$.*

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

- Let $s \rightarrow t$ be due to congruence closure. Then, $\Gamma \vdash_{\mathcal{W}} t : A$ follows by the induction hypothesis. For example, if $s = s_1 s_2, t = t_1 s_2$ and $s_1 \rightarrow t_1$, then we know that $\Gamma \vdash_{\mathcal{W}} s_1 : B \rightarrow A$ and $\Gamma \vdash_{\mathcal{W}} s_2 : B$ for some type B . By induction hypothesis, we then have that $\Gamma \vdash_{\mathcal{W}} t_1 : B \rightarrow A$ and thus $\Gamma \vdash_{\mathcal{W}} t : A$.
- Let $\text{delay}(K[\text{adv } t]) \rightarrow \text{let } x = t \text{ in } \text{delay}(K[\text{adv } x])$ and $\Gamma \vdash_{\mathcal{W}} \text{delay}(K[\text{adv } t]) : A$. That is, $A = \bigcirc A'$ and $\Gamma, \checkmark \vdash_{\mathcal{W}} K[\text{adv } t] : A'$. Then by Lemma A.2, we obtain some type B such that $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark \vdash_{\mathcal{W}} K[\text{adv } x] : A'$. Hence, $\Gamma \vdash_{\mathcal{W}} \text{let } x = t \text{ in } \text{delay}(K[\text{adv } x]) : A$.
- Let $\lambda x.K[\text{adv } t] \rightarrow \text{let } y = \text{adv } t \text{ in } \lambda x.(K[y])$ and $\Gamma \vdash_{\mathcal{W}} \lambda x.K[\text{adv } t] : A$. Hence, $A = A_1 \rightarrow A_2$ and $\Gamma, x : A_1 \vdash_{\mathcal{W}} K[\text{adv } t] : A_2$. Then, by Lemma A.3, there is some type B such that $\Gamma \vdash_{\mathcal{W}} \text{adv } t : B$ and $\Gamma, y : B, x : A_1 \vdash_{\mathcal{W}} K[y] : A_2$. Hence, $\Gamma \vdash_{\mathcal{W}} \text{let } y = \text{adv } t \text{ in } \lambda x.(K[y]) : A$. ■

A.2 Exhaustiveness

Secondly, we show that any closed $\lambda_{\mathcal{W}}$ term that cannot be rewritten any further is also a closed λ_{\checkmark} term (cf. Proposition A.9 below).

Definition A.5.

- (i) We say that a term t is *weakly adv-free* iff whenever $t = K[\text{adv } s]$ for some K and s , then s is a variable.

(ii) We say that a term t is *strictly adv-free* iff there are no K and s such that $t = K[\text{adv } s]$.

Clearly, any strictly adv-free term is also weakly adv-free.

In the following, we use the notation $t \dashrightarrow$ to denote the fact that there is no term t' with $t \longrightarrow t'$; in other words, t is a normal form.

Lemma A.6.

- (i) If $\text{delay } t \dashrightarrow$, then t is weakly adv-free.
- (ii) If $\lambda x. t \dashrightarrow$, then t is strictly adv-free.

Proof Immediate, by the definition of weakly/strictly adv-free and \dashrightarrow . ■

Lemma A.7. Let Γ' contain at least one tick, $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : A, t \dashrightarrow$, and t weakly adv-free. Then $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} t : A$

Proof We proceed by induction on $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : A$:

- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{adv } t : A$: Then there are Γ_1, Γ_2 such that Γ_2 tick-free, $\Gamma' = \Gamma_1, \checkmark, \Gamma_2$, and $\Gamma, \checkmark, \Gamma_1 \vdash_{\checkmark} t : \bigcirc A$. Since $\text{adv } t$ is by assumption weakly adv-free, we know that t is some variable x . Since $\bigcirc A$ is not stable we thus know that $x : \bigcirc A \in \Gamma_1$. Hence, $\Gamma^{\square}, \Gamma_1 \vdash_{\checkmark} t : \bigcirc A$, and therefore $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{adv } t : A$.
- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{delay } t : \bigcirc A$: Hence, $\Gamma, \checkmark, \Gamma', \checkmark \vdash_{\checkmark} t : A$. Moreover, since $\text{delay } t \dashrightarrow$, we have by Lemma A.6 that t is weakly adv-free. We may thus apply the induction hypothesis to obtain that $\Gamma^{\square}, \Gamma', \checkmark \vdash_{\checkmark} t : A$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{delay } t : \bigcirc A$.
- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{box } t : \square A$: Hence, $(\Gamma, \checkmark, \Gamma')^{\square} \vdash_{\checkmark} t : A$, which is the same as $(\Gamma^{\square}, \Gamma')^{\square} \vdash_{\checkmark} t : A$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{box } t : \square A$.
- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \lambda x.t : A \rightarrow B$: That is, $\Gamma, \checkmark, \Gamma', x : A \vdash_{\checkmark} t : B$. Since, by assumption $\lambda x.t \dashrightarrow$, we know by Lemma A.6 that t is strictly adv-free, and thus also weakly adv-free. Hence, we may apply the induction hypothesis to obtain that $\Gamma^{\square}, \Gamma', x : A \vdash_{\checkmark} t : B$, which in turn implies $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \lambda x.t : A \rightarrow B$.
- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{fix } x.t : A$: Hence, $(\Gamma, \checkmark, \Gamma')^{\square}, x : \square \bigcirc A \vdash_{\checkmark} t : A$, which is the same as $(\Gamma^{\square}, \Gamma')^{\square}, x : \square \bigcirc A \vdash_{\checkmark} t : A$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{fix } x.t : A$.
- $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} x : A$: Then, either $\Gamma' \vdash_{\checkmark} x : A$ or $\Gamma^{\square} \vdash_{\checkmark} x : A$. In either case, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} x : A$ follows.
- The remaining cases follow by the induction hypothesis in a straightforward manner. For example, if $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} st : A$, then there is some type B with $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} s : B \rightarrow A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : B$. Since st is weakly adv-free, so are s and t , and we may apply the induction hypothesis to obtain that $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} s : B \rightarrow A$ and $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} t : B$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} st : A$. ■

Lemma A.8. Let $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : A, t \dashrightarrow$ and t strictly adv-free. Then $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} t : A$. (Note that this Lemma is about λ_{\checkmark} .)

Proof We proceed by induction on $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : A$.

- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{adv } t : A}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{adv } t : A}$: Impossible since $\text{adv } t$ is not strictly adv-free.
- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{delay } t : \bigcirc A}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{delay } t : \bigcirc A}$: Hence, $\Gamma^{\square}, \Gamma', \checkmark \vdash_{\checkmark} t : A$, which in turn implies that $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{delay } t : \bigcirc A$.
- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{box } t : \square A}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{box } t : \square A}$: Hence, $(\Gamma, \checkmark, \Gamma')^{\square} \vdash_{\checkmark} t : A$, which is the same as $(\Gamma^{\square}, \Gamma')^{\square} \vdash_{\checkmark} t : A$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{box } t : \square A$.
- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \lambda x.t : A \rightarrow B}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \lambda x.t : A \rightarrow B}$: That is, $\Gamma, \checkmark, \Gamma', x : A \vdash_{\checkmark} t : B$. Since, by assumption $\lambda x.t \dashrightarrow$, we know by Lemma A.6 that t is strictly adv-free. Hence, we may apply the induction hypothesis to obtain that $\Gamma^{\square}, \Gamma', x : A \vdash_{\checkmark} t : B$, which in turn implies $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \lambda x.t : A \rightarrow B$.
- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{fix } x.t : A}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} \text{fix } x.t : A}$: Hence, $(\Gamma, \checkmark, \Gamma')^{\square}, x : \square \bigcirc A \vdash_{\checkmark} t : A$, which is the same as $(\Gamma^{\square}, \Gamma')^{\square}, x : \square \bigcirc A \vdash_{\checkmark} t : A$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} \text{fix } x.t : A$.
- $\frac{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} x : A}{\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} x : A}$: Then, either $\Gamma' \vdash_{\checkmark} x : A$ or $\Gamma^{\square} \vdash_{\checkmark} x : A$. In either case, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} x : A$ follows.
- The remaining cases follow by the induction hypothesis in a straightforward manner. For example, if $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} st : A$, then there is some type B with $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} s : B \rightarrow A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : B$. Since st is strictly adv-free, so are s and t , and we may apply the induction hypothesis to obtain that $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} s : B \rightarrow A$ and $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} t : B$. Hence, $\Gamma^{\square}, \Gamma' \vdash_{\checkmark} st : A$. ■

In order for the induction argument to go through, we generalise the exhaustiveness property from closed λ_{\checkmark} terms to open λ_{\checkmark} terms in a context with at most one tick.

Proposition A.9 (exhaustiveness). *Let $\Gamma \vdash_{\checkmark}, \Gamma \vdash_{\checkmark} t : A$ and $t \dashrightarrow$. Then $\Gamma \vdash_{\checkmark} t : A$.*

Proof We proceed by induction on the structure of t and by case distinction on the last typing rule in the derivation of $\Gamma \vdash_{\checkmark} t : A$.

- $\frac{\Gamma, \checkmark \vdash_{\checkmark} t : A}{\Gamma \vdash_{\checkmark} \text{delay } t : \bigcirc A}$:
We consider two cases:
 - Γ is tick-free: Hence, $\Gamma, \checkmark \vdash_{\checkmark}$ and thus $\Gamma, \checkmark \vdash_{\checkmark} t : A$ by induction hypothesis. Since $|\Gamma| = \Gamma$, we thus have that $\Gamma \vdash_{\checkmark} \text{delay } t : \bigcirc A$.
 - $\Gamma = \Gamma_1, \checkmark, \Gamma_2$ and Γ_2 tick-free: By definition, $t \dashrightarrow$ and, by Lemma A.6, t is weakly adv-free. Hence, by Lemma A.7, we have that $\Gamma_1^{\square}, \Gamma_2, \checkmark \vdash_{\checkmark} t : A$. We can thus apply the induction hypothesis to obtain that $\Gamma_1^{\square}, \Gamma_2, \checkmark \vdash_{\checkmark} t : A$. Since $|\Gamma| = \Gamma_1^{\square}, \Gamma_2$, we thus have that $\Gamma \vdash_{\checkmark} \text{delay } t : \bigcirc A$.
- $\frac{\Gamma, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \lambda x.t : A \rightarrow B}$:
By induction hypothesis, we have that $\Gamma, x : A \vdash_{\checkmark} t : B$. There are two cases to consider:

- Γ is tick-free: Then, $|\Gamma| = \Gamma$ and we thus obtain that $\Gamma \vdash_{\checkmark} \lambda x.t : A \rightarrow B$.
- $\Gamma = \Gamma_1, \checkmark, \Gamma_2$ with Γ_1, Γ_2 tick-free: Since $t \dashrightarrow$ by definition and t strictly adv-free by Lemma A.6, we may apply Lemma A.8 to obtain that $\Gamma_1^\square, \Gamma_2, x : A \vdash_{\checkmark} t : B$. Since $|\Gamma| = \Gamma_1^\square, \Gamma_2$, we thus have that $\Gamma \vdash_{\checkmark} \lambda x.t : A \rightarrow B$.
- All remaining cases follow immediately from the induction hypothesis, because all other rules are either the same for both calculi or the λ_{\checkmark} typing rule has an additional side condition that Γ have at most one tick, which holds by assumption. ■

A.3 Strong normalisation

Proposition A.10 (strong normalisation). *The rewrite relation \longrightarrow is strongly normalising.*

Proof To show that \longrightarrow is strongly normalising, we define for each term t a natural number $d(t)$ such that, whenever $t \longrightarrow t'$, then $d(t) > d(t')$. A *redex* is a term of the form $\text{delay } K[\text{adv } t]$ with t not a variable, or a term of the form $\lambda x.K[\text{adv } s]$. For each redex occurrence in a term t , we can calculate its *depth* as the length of the unique path that goes from the root of the abstract syntax tree of t to the occurrence of the redex. Define $d(t)$ as the sum of the depth of all redex occurrences in t . Since each rewrite step $t \longrightarrow t'$ removes a redex or replaces a redex with a new redex at a strictly smaller depth, we have that $d(t) > d(t')$. ■

Theorem A.11. *For each $\vdash_{\checkmark} t : A$, we can effectively construct a term t' with $t \longrightarrow^* t'$ and $\vdash_{\checkmark} t' : A$.*

Proof We construct t' from t by repeatedly applying the rewrite rules of \longrightarrow . By Proposition A.10, this procedure will terminate and we obtain a term t' with $t \longrightarrow^* t'$ and $t' \dashrightarrow$. According to Proposition A.4, this means that $\vdash_{\checkmark} t' : A$, which in turn implies by Proposition A.9 that $\vdash_{\checkmark} t' : A$. ■

B Coq formalisation

The supplementary material for this article contains a Coq formalisation of our meta-theoretical results. Below we give a high-level overview of the structure of the formalisation.

Main results. The main results are proved in the following three Coq files:

- `FundamentalProperty.v` proves Theorem 5.1, the fundamental property of the logical relation.
- `Productivity.v` proves Theorem 4.2, that is, the stream semantics is safe.
- `Causality.v` proves Theorem 4.3, that is, the stream transducer semantics is safe.

Language definition. The following files define the λ_{\checkmark} calculus and prove auxiliary lemmas:

- `RawSyntax.v` defines the syntax of λ_{\checkmark} .
- `Substitutions.v` defines substitutions of types and terms.
- `Typing.v` defines the type system of λ_{\checkmark} .
- `Reduction.v` defines the big-step operational semantics.
- `Streams.v` defines the small-step operational semantics for streams and stream transducers.

Logical relation. The following files contain the central definitions and lemmas for the logical relation argument:

- `Heaps.v` defines heaps.
- `Worlds.v` defines stores and the various orderings that are used in the logical relation.
- `LogicalRelation.v` defines the logical relation and proves characterisation lemmas that correspond to the definition of the logical relation in Figure 9.
- `LogicalRelationAux.v` proves lemmas about the logical relation that are used in the proof of the fundamental property.

Further details about the formalisation can be found in the accompanying README file.