Modal FRP For All

Functional Reactive Programming Without Space Leaks in Haskell

PATRICK BAHR, IT University of Copenhagen, Denmark

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 productive, 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 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. This 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. All proofs have been formalised using the Coq proof assistant.

Additional Key Words and Phrases: Functional reactive programming, Modal types, Haskell, Type systems

1 INTRODUCTION

Reactive systems perform an ongoing interaction with their environment, receiving inputs from the outside, changing their internal state and producing some output. Examples of such systems include GUIs, web applications, video games, and robots. Programming such systems with traditional general-purpose imperative languages can be very 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 difficult and error-prone.

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, thus providing a rich and expressive programming model. In addition, we can easily 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 leads to the question of what the interface for signals and events should be. A naive approach would be to model signals as streams defined by the following Haskell data type[1]

$$\textbf{data}\ \mathit{Str}\ a = a ::: (\mathit{Str}\ a)$$

which encodes a stream of type *Str a* as 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.

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

Author's address: Patrick Bahr, IT University of Copenhagen, Denmark, paba@itu.dk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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:

$$\begin{aligned}&\mathit{map} :: (a \to b) \to \mathit{Str}\ a \to \mathit{Str}\ b\\&\mathit{map}\ f\ (x ::: xs) = f\ x ::: \mathit{map}\ f\ xs\end{aligned}$$

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

$$\begin{aligned}&\mathit{clairvoyance} :: \mathit{Str}\ \mathit{Int} \to \mathit{Str}\ \mathit{Int}\\&\mathit{clairvoyance}\ (x ::: xs) = \mathit{map}\ (+1)\ xs\end{aligned}$$

This function takes the input n of the *next* time step and returns $n + 1$ 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 avoiding 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.

In recent years, an alternative to this approach has been developed [Bahr et al. 2019; Jeffrey 2014; Jeltsch 2013; Krishnaswami 2013; Krishnaswami and Benton 2011; Krishnaswami et al. 2012] that uses a *modal* type operator $\bigcirc$ that 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:

$$\textbf{data}\ \mathit{Str}\ a = a ::: (\bigcirc(\mathit{Str}\ a))$$

That is, a stream of type $\mathit{Str}\ a$ is an element of type a now and a stream of type $\mathit{Str}\ a$ later, thus separating each element 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 $(\bigcirc a \to a) \to a$ gives a powerful type system for reactive programming that guarantees not only causality, but also *productivity*, i.e. 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 can be very difficult to reason about their space and time cost. A reactive program that exhibits a gradually slower response time, i.e. 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, e.g. if it holds on to memory while continually allocating more.

In recent years, there has been an effort to devise FRP languages that avoid *implicit* space leaks, i.e. space leaks that are caused by the implementation of the FRP language rather than explicit memory allocations intended by the programmer. For example, 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 permit an aggressive garbage collection strategy that rules out implicit space leaks. Moreover, Krishnaswami proved this memory property using a novel proof technique based on logical relations.

The absence of space leaks is an operational property that is notoriously difficult to reason about in higher-level languages. For example, consider the following innocuously looking function *const* that takes an element of type a and repeats it indefinitely as a stream:

$$
\begin{aligned}
&\mathit{const} :: a \longrightarrow \mathit{Str}\ a \\
&\mathit{const}\ x = x ::: \mathit{const}\ x
\end{aligned}
$$

In particular, this function can be instantiated at type $\mathit{const} :: \mathit{Str}\ \mathit{Int} \longrightarrow \mathit{Str}\ (\mathit{Str}\ \mathit{Int})$, which has an inherent space leak with its memory usage growing linearly with time: At each time step n it has to store all previously observed input values from time step 0 to n. On the other hand, instantiated with the type $\mathit{const} :: \mathit{Int} \longrightarrow \mathit{Str}\ \mathit{Int}$, the function can be efficiently implemented. To distinguish between these two scenarios, Krishnaswami [2013] introduced the notion of *stable types*, i.e. types such as *Int* that are time invariant and whose values can thus be transported into the future without causing space leaks.

Contributions. In this paper, we present Rattus, a practical modal FRP language based on the modal FRP calculi of Krishnaswami [2013] and Bahr et al. [2019] but with a simpler 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] to avoid the syntactic overhead of the dual-context-style type system of Krishnaswami [2013]. But we simplify the typing system by reducing the number of *tokens* (from two down to one), extending the language's expressivity, and simplifying the guarded recursion scheme. Despite its simpler type system it retains the operational guarantees of these 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 similar to Krishnaswami's, and we have formalised the proof using the Coq theorem prover (see supplementary material).

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 our language and a plugin for the GHC Haskell compiler. The latter is necessary to check the more restrictive variable scope rules of Rattus and to ensure an eager evaluation strategy that is central to 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 arrowized FRP library in the style of Yampa. We then use these FRP libraries to implement a primitive game. 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 through examples. Section 3 presents a case study of a simple FRP library based on streams and events, as well as an arrowized 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. 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 RATTUS NORVEGICUS DOMESTICA

2.1 Delayed computations

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:

$\textbf{data}\ \mathit{Str}\ a = a ::: (\bigcirc(\mathit{Str}\ a))$

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

$\mathit{const} :: \mathit{Int} \longrightarrow \mathit{Str}\ \mathit{Int}$
$\mathit{const}\ x = x ::: \mathsf{delay}\ (\mathit{const}\ x)$

Because the tail of a stream of integers must be of type $\bigcirc(\mathit{Str}\ \mathit{Int})$, we have to use delay, which is the introduction form for the type modality $\bigcirc$. Intuitively speaking, delay moves a computation one time step into the future. We could think of delay having type $a \longrightarrow \bigcirc a$, 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 rules for delay is formulated as follows:

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

This is a characteristic example of a Fitch-style typing rule: It introduces the *token* ✓ (pronounced “tick”) in the typing context Γ. A typing context consists of type assignments of the form $x :: A$ but it can also contain *at most one* such token ✓. We can think of ✓ as denoting the passage of one time step, i.e. all variables to the left of ✓ 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 2.2.

Formally, the variable introduction rule of Rattus is 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 ✓, then it is no longer in scope.

Turning back to our definition of the *const* function, we can see that the recursive call *const x* must be of type *Str Int* in the context Γ, ✓, where Γ contains $x :: \mathit{Int}$. 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 *const* to arbitrary types:

$\mathit{leakyConst} :: a \longrightarrow \mathit{Str}\ a$
$\mathit{leakyConst}\ x = x ::: \mathsf{delay}\ (\mathit{leakyConst}\ x)$ -- 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 ✓ introduced by delay.

The definition of *const* 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 ✓ – 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
```

Here the recursive call *loop* does not occur under a delay, and thus would be rejected by the type checker.

The function *inc* below takes a stream of integers as input and increments each integer by 1:

```
inc :: Str Int → 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 $\bigcirc(\mathit{Str}\ \mathit{Int})$ into type $\mathit{Str}\ \mathit{Int}$. Again we could think of adv having type $\bigcirc a \to a$, but this general type would allow us to write non-causal functions such as the following:

```
tomorrow :: Str Int → Str Int
tomorrow (x ::: xs) = adv xs   -- adv is not allowed here
```

This function skips one time step 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 ✓:

$$\frac{\Gamma \vdash t :: \bigcirc A}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\ t :: A}$$

Not only does adv require a ✓, it also causes all bound variables to the right of ✓ 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.

In summary, the typing context can be of two different forms: either Γ with no ✓, or of the form $\Gamma, \checkmark, \Gamma'$ with exactly one tick. The former means that we are programming in the present, whereas the latter means we are programming one time step into the future where Γ' contains variables bound one time step after the variables in Γ. We can move between these two forms by delay and adv. Moreover, the ✓ 'hides' non-stable variables as expressed in the variable typing rule. So in the future we do not have access to non-stable variables from the past.

2.2 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. The intuition here is that $\bigcirc$ 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.

However, that means we cannot not implement the *map* function that takes a function $f :: a \to b$ and applies it to each element of a stream of type $\mathit{Str}\ a$, because it would require us to apply the function f at any time in the future. We cannot do this because $a \to b$ is not a stable type and therefore f cannot be transported into the future. However, Rattus has the type modality $\Box$, pronounced "box", that turns any type A into a stable type $\Box A$. Using the $\Box$ modality we can implement *map* as follows:

```
map :: □(a → b) → Str a → Str b
map f (x ::: xs) = unbox f x ::: delay (map f (adv xs))
```

Instead of a function of type $a \to b$, *map* takes a *boxed* function f of type $\Box(a \to b)$ as 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 has simply type $\Box a \to a$, this time without any side conditions.

On the other hand, the corresponding introduction form for □ has to make sure that boxed values do not refer to non-stable variables:

$$\frac{\Gamma^{\square} \vdash t :: A}{\Gamma \vdash \mathsf{box}\ t :: \square A}$$

Here, $\Gamma^{\square}$ denotes the typing context that is obtained from Γ by removing all non-stable types and the ✓ token if present:

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

Thus, for a well-typed term 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 → Str Int
inc = map (box (+1))
```

Using the □ modality we can also generalise the constant stream function to arbitrary boxed types:

```
constBox :: □a → 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 ⇒ a → Str a
const x = x ::: delay (const x)
```

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* that recurses over the stream:

```
map :: □(a → b) → Str a → Str b
map f = run
  where run :: Str a → Str b
        run (x ::: xs) = unbox f x ::: delay (run (adv xs))
```

Here *run* is type checked in a typing environment Γ that contains $f :: \square(a \to b)$. Since *run* is defined by guarded recursion, we require that its definition must type check in the typing context $\Gamma^{\square}$. Because f is of a stable type, it remains in $\Gamma^{\square}$ and is thus in scope in the definition of *run*. So guarded recursive definitions interact with the typing environment in the same way as box. That way, we are sure that the recursive definition is 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 → b) → Str a → Str b
leakyMap f = run
  where run :: Str a → 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 ✓ 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.

2.3 Ruling out implicit space leaks

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 and productive. Furthermore, the typing rules also ensure that Rattus has no implicit space leaks. In simple terms, this means that temporal values, i.e. 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.

In order to rule out space leaks, the type system imposes restrictions on which computations and data we can move into the future. In particular, we have to be very careful with function types since closures can implicitly store arbitrary data. This observation is also the reason why function types are not considered stable. If function types were considered stable, we could implicitly transport arbitrary data across time and thus cause space leaks.

In addition, we must restrict where function definitions may appear. They are not allowed in the context of a ✓:

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

Indeed Bahr et al. [2019] gave a counterexample that shows that allowing ✓ in lambda abstractions would break the safety of their operational semantics that ensures the absence of implicit space leaks in their Simply RaTT calculus. The counterexample also applies here and would cause space leaks in Rattus.

In practice, we have not found the above restriction to impose any limitation on the programmer. We may still allow functions to be defined in the context of a ✓, but then the body of the function must typecheck without the ✓ and without the non-stable variables that occurred to the left of that ✓:

$$\frac{\Gamma^{\Box}, \Gamma', x :: A \vdash t :: B}{\Gamma, \checkmark, \Gamma' \vdash \lambda x \to t :: A \to B}$$

However, we have yet to find a practical example where we need to define a function under a ✓.

To achieve the goal of ruling out space leaks, we have to be careful about the evaluation strategy as well. Generally speaking, we need to evaluate as soon as possible but delay computations whose result are only needed in the next time step. In other words, Rattus programs are executed using a call-by-value semantics, except for delay and box. That is, arguments are evaluated to values before they are passed on to functions. This is made more precise in section 4. In the Haskell embedding of the language, this evaluation strategy is enforced by using strict data structures and strict evaluation. The latter is achieved by a compiler plug-in that transforms all Rattus functions so that arguments are always evaluated to weak head normal form (cf. section 6).

3 REACTIVE PROGRAMMING IN RATTUS

3.1 Programming with streams and events

In this section we showcase how Rattus can be used for reactive programming. To this end we use a small library of combinators for programming with streams and events defined in Figure 1.

The *map* function should be familiar by now. The *zip* function combines to streams similar to Haskell's *zip* function on lists. Note however that instead of the normal pair type we use a strict pair type:

```
map :: □(a → b) → Str a → Str b
map f (x ::: xs) = unbox f x ::: delay (map f (adv xs))

zip :: Str a → Str b → Str (a ⊗ b)
zip (a ::: as) (b ::: bs) = (a ⊗ b) ::: delay (zip (adv as) (adv bs))

scan :: Stable b ⇒ □(b → a → b) → b → Str a → Str b
scan f acc (a ::: as) = acc′ ::: delay (scan f acc′ (adv as))
  where acc′ = unbox f acc a

type Events a = Str (Maybe′ a)

switch :: Str a → Events (Str a) → Str a
switch (x ::: xs) (Nothing′         ::: fas) = x ::: (delay switch ⊛ xs ⊛ fas)
switch _          (Just′ (a ::: as) ::: fas) = a ::: (delay switch ⊛ as ⊛ fas)

switchTrans :: (Str a → Str b) → Events (Str a → Str b) → (Str a → Str b)
switchTrans f es as = switchTrans′ (f as) es as

switchTrans′ :: Str b → Events (Str a → Str b) → Str a → Str b
switchTrans′ (b ::: bs) (Nothing′ ::: fs) as = b  ::: (delay switchTrans′ ⊛ bs  ⊛ fs ⊛ tail as)
switchTrans′ _          (Just′ f  ::: fs) as = b′ ::: (delay switchTrans′ ⊛ bs′ ⊛ fs ⊛ tail as)
  where (b′ ::: bs′) = f as
```

Fig. 1. Small library for streams and events.

```
data a ⊗ b = !a ⊗ !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 $v_0, v_1, v_2, \ldots$, the expression *scan l* (box f) v computes the stream

$$f\ v\ v_0, f\ (f\ v\ v_0)\ v_1, f\ (f\ (f\ v\ v_0)\ v_1)\ v_2, \ldots$$

If one would want a variant of *scan* that is closer to Haskell's *scanl*, i.e. 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 functional reactive programming is that it must provide a way to react to events. In particular, it must support the ability to *switch* behaviour as reaction to the occurrence of an event. There are different ways to represent events. The simplest is to define 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 *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. Note that the behaviour changes *every time* an event occurs, not only the first time.

In the definition of *switch* we use the applicative operator ⊛ defined as follows

```
(⊛) :: ◯(a → b) → ◯a → ◯b
f ⊛ x = delay ((adv f) (adv x))
```

Instead of using ⊛, we could have also written delay (*switch* (adv *xs*) (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′* where the initial stream function is rather just a stream.

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 ⊗ 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):

```
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 keep track of some internal state, e.g. the position and velocity of the ball, while consuming the input stream. The internal state is then projected away 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 towards the upper right corner.

Let's change the implementation of *pong* so that it allows the player to reset the game, e.g. after ball has fallen off the screen:

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

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

```
class Category a ⇒ Arrow a where
  arr    :: (b → c) → a b c
  first  :: a b c → a (b, d) (c, d)
  second :: a b c → a (d, b) (d, c)
  (***)  :: a b c → a b′ c′ → a (b, b′) (c, c′)
  (&&&)  :: a b c → a b c′ → a b (c, c′)
```

```
class Category cat where
  id   :: cat a a
  (∘)  :: cat b c → cat a b → cat a c

class Arrow a ⇒ ArrowLoop a where
  loop :: a (b, d) (c, d) → a b c
```

Fig. 2. Arrow type class.

```
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 *Events* (*Str* (*Float* ⊗ *Input*) → *Str Pos*), which will be triggered every time the player hits the reset button.

3.3 Arrowized FRP

The benefit of a modal FRP language is that we can directly interact with signals and events without giving up on causality. A popular alternative to ensure causality is arrowized FRP [Nilsson et al. 2002], which takes *signal functions* as primitive and uses Haskell's arrow notation [Paterson 2001] to construct them. But Rattus promises more than just causality, it also ensures productivity and avoids implicit space leaks. That means, there is merit in implementing an arrowized FRP interface in Rattus.

At the centre of arrowized 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)
```

We can almost copy the definition of *SF* from Nilsson et al. [2002], but we have to insert the ◯ modality to make it a guarded recursive type:

```
data SF a b = SF (Float → a → (◯(SF a b), b))
```

Implementing the methods of the *Arrow* type class is straightforward except for 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))
```

In other words, the *arr* method is a potential source for space leaks in the implementation of arrowized FRP. To avoid this, we have to give it the above more restrictive type.

But fortunately, that does not stop our effort in using the arrow notation. By treating *arr f* as a short hand for *arrBox* (box *f*) Haskell will still allow us to use the arrow notation while Rattus makes sure that box *f* is still well-typed, i.e. *f* only refers to variables of stable type.

There are a number of other combinators that we need to provide to program with signal functions, such as combinators 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:

```
pong′ :: SF Input (Pos ⊗ Float)
pong′ = 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 provide a very general recursion principle, the *loop* method of the *ArrowLoop* class in Figure 2. We cannot implement *loop* using guarded recursion. However, Yampa also provides a more rigid combinator *loopPre*, which we can implement:

```
loopPre :: c → SF (a, c) (b, ◯c) → SF a b
loopPre c (SF sf) = SF (λd a → let (r, (b, c′)) = sf d (a, c)
                               in (delay (loopPre (adv c′) (adv r)), b))
```

Apart from the addition of the ◯ modality, this definition has the same type as Yampa's.

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

```
ballPos :: SF (Float ⊗ Input) Pos
ballPos = loopPre (20 ⊗ 50) run where
  run :: SF ((Float ⊗ Input), Vel) (Pos, ◯Vel)
  run = proc ((pad ⊗ inp), v) → do p ← integral (0 ⊗ 0) ─≺ v
                                   returnA ─≺(p, delay (calculateNewVelocity 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:

```
integral :: (Stable a, VectorSpace a s) ⇒ a → SF a a
integral acc = SF (λt a → let acc′ = acc ^+^ (realToFrac t *^ a)
                          in (delay (integral acc′), acc′))
```

This combinator works on any type *a* that implements the *VectorSpace* type class providing a vector addition operator ^+^ and a scalar multiplication operator *^.

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.

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

Fig. 3. Syntax of (stable) types, terms, and values. In typing rules, only closed types (no free α) are considered.

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

Fig. 4. Well-formed contexts

4 CORE CALCULUS

In this section we present the core calculus of Rattus. The purpose of this calculus is to formally present the language's Fitch-style typing rules, its operational semantics, and to formally prove the central operational properties, i.e. productivity, causality, and absence of implicit space leaks. To this end, the calculus is stripped down to its essence: simply typed lambda calculus extended with guarded recursive types $\mathsf{Fix}\, \alpha.A$ and the two type modalities $\Box$ 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 the core calculus. Besides guarded recursive types and the two type modalities, we include standard sum and product types along with unit and integer types. The type of streams of type A would be represent as $\mathsf{Fix}\, \alpha.A \times \alpha$. Note the absence of $\bigcirc$ in this type. When unfolding guarded recursive types such as $\mathsf{Fix}\, \alpha.A \times \alpha$, the $\bigcirc$ modality is inserted implicitly: $\mathsf{Fix}\, \alpha.A \times \alpha \cong A \times \bigcirc(\mathsf{Fix}\, \alpha.A \times \alpha)$. This ensures that guarded recursive types are by construction always guarded by the $\bigcirc$ modality.

Typing contexts, defined in Figure 4, consist of variable typings $x : A$ and may contain at most one $\checkmark$ token. If a typing context contains no $\checkmark$, we call it *tick-free*. The complete set of typing rules for the core calculus are given in Figure 5. The typing rules that we have presented for the surface language in section 2 appear in the same form also here, except for the change of 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, the rule enforces the body t of the fixed point to be stable by strengthening the typing context to $\Gamma^{\Box}$. To see how the recursion syntax of the surface language translates into the fixed point combinator, let us reconsider the *const* function:

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

Such a recursive definition is simply translated into a fixed point $\mathsf{fix}\, r.t$ where the recursive occurrence of *const* is replaced by $\mathsf{adv}\ r$.

$$
const = \mathsf{fix}\, r.\lambda x.x ::: \mathsf{delay}(\mathsf{adv}\, r\, x)
$$

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

$$\frac{\Gamma \vdash s : \mathsf{Int} \qquad \Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash s + t : \mathsf{Int}} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \text{ tick-free}}{\Gamma \vdash \lambda x.t : A \longrightarrow B} \qquad \frac{\Gamma^{\square}, \Gamma', x : A \vdash t : B}{\Gamma, \checkmark, \Gamma' \vdash \lambda x.t : A \longrightarrow B}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \mathsf{let}\, x \,=\, s \,\mathsf{in}\, t : B} \qquad \frac{\Gamma \vdash t : A \longrightarrow B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B}$$

$$\frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, t : A_i} \qquad \frac{\Gamma \vdash t : A_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2}$$

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

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

$$\frac{\Gamma \vdash t : A[\bigcirc(\mathsf{Fix}\, \alpha.A)/\alpha]}{\Gamma \vdash \mathsf{into}\, t : \mathsf{Fix}\, \alpha.A} \qquad \frac{\Gamma \vdash t : \mathsf{Fix}\, \alpha.A}{\Gamma \vdash \mathsf{out}\, t : A[\bigcirc(\mathsf{Fix}\, \alpha.A)/\alpha]} \qquad \frac{\Gamma^{\square}, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : A}$$

Fig. 5. Typing rules.

where the stream cons operator $s ::: t$ is shorthand for into $\langle s, t \rangle$. The variable r is of type $\bigcirc(Int \longrightarrow Str\ Int)$ and applying adv turns it into type $Int \longrightarrow Str\ Int$. Moreover, the restriction that recursive calls must occur in a context with ✓ makes sure that this transformation from recursion notation to fixed point combinator is type-preserving.

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

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

```
leakyMap :: (a → b) → Str a → Str b
leakyMap f = run
  where run :: Str a → Str b
        run (x ::: xs) = f x ::: delay (leakyMap (adv xs))
```

Translated into the core calculus, it looks like this:

$$leakyMap = \lambda f.\mathsf{fix}\, r.\lambda s.f\,(\mathsf{head}\, s) ::: \mathsf{delay}((\mathsf{adv}\, r)\ (\mathsf{adv}(\mathsf{tail}\, s)))$$

Here the pattern matching syntax is translated into projection functions head and tail that decompose a stream into its head and tail, respectively. 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.

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle \overline{m};\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle \overline{n};\sigma''\rangle}{\langle t+t';\sigma\rangle \Downarrow \left\langle \overline{m+n};\sigma''\right\rangle}$$

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

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

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

$$\frac{l=\mathsf{alloc}\,(\sigma)}{\langle \mathsf{delay}\,t;\sigma\rangle \Downarrow \langle l;\sigma,l\mapsto t\rangle} \qquad \frac{\langle t;\eta_N\rangle \Downarrow \left\langle l;\eta'_N\right\rangle \qquad \left\langle \eta'_N(l);\eta'_N\checkmark\eta_L\right\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{adv}\,t;\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\sigma'\rangle}$$

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

$$\frac{\langle t[l/x];\sigma,l\mapsto \mathsf{fix}\,x.t\rangle \Downarrow \langle v;\sigma'\rangle \qquad l=\mathsf{alloc}\,(\sigma)}{\langle \mathsf{fix}\,x.t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

Fig. 6. Evaluation semantics.

$$\frac{\langle t;\eta\checkmark\rangle \Downarrow \langle v ::: l;\eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{adv}\,l;\eta_L\rangle} \qquad \frac{\langle t;\eta,l^*\mapsto v ::: l^*\checkmark l^*\mapsto\langle\rangle\rangle \Downarrow \langle v' ::: l;\eta_N\checkmark\eta_L,l^*\mapsto\langle\rangle\rangle}{\langle t;\eta\rangle \overset{v/v'}{\Longrightarrow} \langle \mathsf{adv}\,l;\eta_L\rangle}$$

Fig. 7. Step semantics for streams.

4.2 Operational Semantics

To prove that Rattus is free of implicit space leaks, we devise an operational semantics that after each time step deletes all data from the previous time step. This characteristics makes the operational semantics *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 can 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 operational semantics consists of two components: the *evaluation semantics*, presented in Figure 6, which describes the operational behaviour of Rattus within a single time step; and the *step semantics*, presented in Figure 7, which describes the behaviour of a program over time, e.g. how it consumes and constructs streams.

The evaluation semantics is given as a 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, i.e. $\sigma = \eta_L$, 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. Given a store σ of the form η_L or $\eta_N \checkmark \eta_L$, we write $\mathsf{alloc}\,(\sigma)$ for 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 $\eta'_L = \eta_L, l \mapsto t$, i.e. 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 on the later heap and returns its location on the heap. Conversely, if we apply adv to such a delayed computation, we retrieve the term from the now heap and evaluate it.

Also the guarded fixed point combinator fix allocates a delayed computation on the store. In a term $\mathsf{fix}\,x.t$ of type A, variable x has type $\bigcirc A$. So when evaluating $\mathsf{fix}\,x.t$ we substitute $\mathsf{delay}(\mathsf{fix}\,x.t)$ for x in t. But since Rattus is a call-by-value language we first evaluate $\mathsf{delay}(\mathsf{fix}\,x.t)$ to a value before substitution. Hence, the operational semantics for $\mathsf{fix}\,x.t$ substitutes the heap location l that points to the delayed computation $\mathsf{fix}\,x.t$.

4.3 Main results

The step semantics 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* $\longrightarrow$ *Str B*. The former just produces an infinite stream of values of type A whereas the latter is reactive process that produces a value of type B for each input value of type A.

4.3.1 Productivity of the step semantics. The small-step semantics $\overset{v}{\Longrightarrow}$ from Figure 7 describes the unfolding of streams of type *Str A*. Given a closed term $\vdash t : Str A$, it produces an infinite reduction sequence

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

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 semantics, we evaluate $\langle t_i; \eta_i \checkmark \rangle \Downarrow \left\langle v_i ::: l; \eta'_i \checkmark \eta_{i+1} \right\rangle$, 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.

As an example consider the following definition of the stream of consecutive numbers starting from some given number:

$$\begin{aligned}&\mathit{from} :: \mathit{Int} \longrightarrow \mathit{Str}\ \mathit{Int}\\ &\mathit{from}\ n = n ::: \mathsf{delay}\ (\mathit{from}\ (n+1))\end{aligned}$$

This definition translates to the following core calculus term:

$$\mathit{from} = \mathsf{fix}\, r.\lambda n.n ::: \mathsf{delay}(\mathsf{adv}\, r\ (n + \bar{1}))$$

Let's see how the stream $\mathit{from}\,\overline{0}$ of type *Str Int* unfolds:

$$\begin{array}{rcl}\left\langle \mathit{from}\,\overline{0};\emptyset\right\rangle & \overset{\overline{0}}{\Longrightarrow} & \left\langle \mathsf{adv}\,l_1';l_1\mapsto \mathit{from},\ l_1'\mapsto \mathsf{adv}\,l_1\,(\overline{0}+\overline{1})\right\rangle\\ & \overset{\overline{1}}{\Longrightarrow} & \left\langle \mathsf{adv}\,l_2';l_2\mapsto \mathit{from},\ l_2'\mapsto \mathsf{adv}\,l_2\,(\overline{1}+\overline{1})\right\rangle\\ & \overset{\overline{2}}{\Longrightarrow} & \left\langle \mathsf{adv}\,l_3';l_3\mapsto \mathit{from},\ l_3'\mapsto \mathsf{adv}\,l_3\,(\overline{2}+\overline{1})\right\rangle\\ & \vdots & \end{array}$$

In each step of the stream unfolding 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 execution of programs 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:

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

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 4.1 (productivity). *Given a term* $\vdash t : \mathsf{Str}\,A$ *with* A *a value type, there is an infinite reduction sequence*

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

such that $\vdash v_i : A$ *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 *Causality of the step semantics.*

The small-step semantics $\overset{v/v'}{\Longrightarrow}$ from Figure 7 describes how a term of type $\mathit{Str}\ A \longrightarrow \mathit{Str}\ B$ transforms a stream of inputs into a stream of outputs in a step-by-step fashion. Given a closed term $\vdash t : \mathsf{Str}\,A \longrightarrow \mathsf{Str}\,B$, and an infinite stream of input values $\vdash v_i : A$, it produces an infinite reduction sequence

$$\langle t;\emptyset\rangle \overset{v_0/v_0'}{\Longrightarrow} \langle t_1;\eta_1\rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_2;\eta_2\rangle \overset{v_2/v_1'}{\Longrightarrow} \dots$$

where each output value v_i' has type B.

The definition of $\overset{v/v'}{\Longrightarrow}$ 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. In each step, we evaluate the current term t_i in the current heap η_i

$$\langle t_i;\eta_i, l^*\mapsto v_i ::: l^* \checkmark l^*\mapsto\langle\rangle\rangle \Downarrow \left\langle v_i' ::: l;\eta_i'\checkmark\eta_{i+1}, l^*\mapsto\langle\rangle\right\rangle$$

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 $v_{i+1} ::: l^*$ 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:

```
sum :: Str Int → Str Int
sum = run 0 where
  run :: Int → Str Int → Str Int
  run acc (x ::: xs) = let acc′ = acc + x
                       in acc′ ::: delay (run acc′ (adv xs))
```

This function definition translates to the following term *sum* in the core calculus, where we use the notation $\mathsf{let}\, x = s\ \mathsf{in}\ t$ for $(\lambda x.t)s$:

$$run = \mathsf{fix}\, r.\lambda acc.\lambda s.\mathsf{let}\, acc' = acc + \mathsf{head}\, s\ \mathsf{in}\ acc' ::: \mathsf{delay}(\mathsf{adv}\, r\, acc'(\mathsf{adv}\,(\mathsf{tail}\, s)))$$

$$sum = run\,\overline{0}$$

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

$$\begin{aligned}
& \langle sum; \emptyset \rangle \\
\overset{\overline{2}/\overline{2}}{\Longrightarrow}\ & \left\langle \mathsf{adv}\, l_1'; l_1 \mapsto run,\ l_1' \mapsto \mathsf{adv}\, l_1\ (\overline{0} + \overline{2})\ (\mathsf{adv}\,(\mathsf{tail}\,(\overline{2} :: l^*))) \right\rangle \\
\overset{\overline{11}/\overline{13}}{\Longrightarrow}\ & \left\langle \mathsf{adv}\, l_2'; l_2 \mapsto run,\ l_2' \mapsto \mathsf{adv}\, l_2\ (\overline{2} + \overline{11})\ (\mathsf{adv}\,(\mathsf{tail}\,(\overline{11} :: l^*))) \right\rangle \\
\overset{\overline{5}/\overline{18}}{\Longrightarrow}\ & \left\langle \mathsf{adv}\, l_3'; l_3 \mapsto run,\ l_3' \mapsto \mathsf{adv}\, l_3\ (\overline{13} + \overline{5})\ (\mathsf{adv}\,(\mathsf{tail}\,(\overline{5} :: l^*))) \right\rangle \\
\vdots\ &
\end{aligned}$$

in each step of the computation the location l_i stores the fixed point *run* and l_i' stores the computation that calls that fixed point with the new accumulator value (0 + 2, 2 + 11, and 13 + 5, 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 4.2 (causality). *Given a term* $\vdash t : \mathsf{Str}\, A \longrightarrow \mathsf{Str}\, B$ *with* B *a value type, and an infinite sequence of values* $\vdash v_i : A$, *there is an infinite reduction sequence*

$$\langle t; \emptyset \rangle \overset{v_0/v_0'}{\Longrightarrow} \langle t_1; \eta_1 \rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_2; \eta_2 \rangle \overset{v_2/v_2'}{\Longrightarrow} \dots$$

such that $\vdash v_i' : B$ *for all* $i \geq 0$.

Since the operational semantics is deterministic, in each step $\langle t_i; \eta_i \rangle \overset{v_i/v_i'}{\Longrightarrow} \langle t_{i+1}; \eta_{i+1} \rangle$ 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 $j > i$.

4.4 Limitations

Now that we have formally precise statements about the operational properties of Rattus, 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 Theorem 4.1 and Theorem 4.2 state that reactive programs in Rattus can be executed effectively – they always make progress and never depend on data that is not yet available. In the Haskell embedding of the language this has to be of course qualified as we can use Haskell functions that loop or crash.

In addition, by virtue of the operational semantics, the two theorems 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 (in Figure 7) discards the 'now' heap and only retains the 'later' heap for the next step.

However, we can still *explicitly* accumulate data and thereby create space leaks. For example, given a strict list type

```
data List a = Nil | !a :! !(List a)
```

we can construct a function that buffers the entire history of an input stream

```
buffer :: Stable a ⇒ Str a → Str (List a)
buffer = scan (box (λxs x → x :! xs)) Nil
```

Given that we have a function *sum* :: *List Int* → *Int* that computes the sum of a list of numbers, we can write the following alternative implementation of the *sums* function using *buffer*:

```
leakySums1 :: Str Int → Str Int
leakySums1 = map (box sum) ∘ 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.

Another example of a time leak is found in the following definition of a stream of all consecutive natural numbers

```
leakyNats :: Str Int
leakyNats = 0 ::: delay (map (box (+1)) leakyNats)
```

The problem here is that this definition computes the nth element of the stream by evaluating $0 + \underbrace{1 + \cdots + 1}_{n \text{ times}}$.[3]

The space leak in *leakySums1* is quite obvious to spot in the explicit allocation of a buffer of type *List Int*. However, these space leaks can be sometimes a bit more subtle when this accumulation of data occurs as part of a closure. We can see this behaviour in the following alternative implementation of the *sums* function that works similarly to the *leakyNats* example above:

```
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. Thus *leakySum′* exhibits the same space and time leak as *leakySum*.

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. The other two examples are more subtle and the leaky behaviour is rooted in a time leak as the programs construct an increasing computation in each step. Below is yet another leaky variant of the *sums* function that explicitly accumulates a computation of type *Int* → *Int* to compute the sum:

```
leakySum3 :: □(Int → Int) → Str Int → Str Int
leakySum3 f (x ::: xs) = unbox f x ::: (delay (leakySum3 (box (λy → unbox f (y + x)))) ⊛ xs)
```

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], although some reformulation is necessary for the latter calculus. For more details we refer to the discussion on related work in section 7.2.

[3] But GHC is quiet clever and will produce efficient code for *leakyNats* anyway.

5 META THEORY

Our goal is to show that Rattus's core calculus enjoys the three central operational properties: productivity, causality and absence of implicit space leaks. These properties are stated in Theorem 4.1 and Theorem 4.2, and we show in this section how these are proved. 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 is fully formalised in the accompanying Coq proofs, we only give a high-level overview of the proof's constructions.

We prove the abovementioned 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 $[\![A]\!]_w$ indexed over some world w with some closure conditions when the index w changes. In our case, $[\![A]\!]_w$ 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 $\overline{\eta}$ 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 $l \in \mathrm{dom}\,(\eta)$. Infinite sequences of heaps are ordered pointwise according to $\sqsubseteq$. Moreover, we extend the ordering to stores in two different ways:

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

That is, $\sqsubseteq$ is the pointwise extension of the order on heaps to stores, and $\sqsubseteq_{\checkmark}$ 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 $\mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}$ and the term relation $\mathcal{T}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}$. Both are defined in Figure 8 by well-founded recursion according to the lexicographic ordering on the triple $(\nu, |A|, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

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

In the definition of the logical relation, we use the notation $\eta; \overline{\eta}$ to denote an infinite sequence of heaps that starts with the heap η and then continues as the sequence $\overline{\eta}$. 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 $\mathrm{gc}(\sigma) \checkmark \eta$. Here $\mathrm{gc}(\sigma)$ denotes the garbage collection of the store σ as defined in Figure 8. It simply drops the 'now' heap if present. To see how this definition captures causality we have to look a the index $\eta; \overline{\eta}$ of future heaps. It changes to the index $\overline{\eta}$, i.e. all future heaps are one time step closer, and the very first future heap η becomes the new 'later' heap in the store index $\mathrm{gc}(\sigma) \checkmark \eta$, whereas the old 'later' heap in σ becomes the new 'now' heap.

$$
\begin{aligned}
\mathcal{V}_\nu[\![\mathsf{Int}]\!]^{\overline{\eta}}_{\sigma} &= \{\overline{n} \mid n \in \mathbb{Z}\},\\
\mathcal{V}_\nu[\![1]\!]^{\overline{\eta}}_{\sigma} &= \{\langle\rangle\},\\
\mathcal{V}_\nu[\![A \times B]\!]^{\overline{\eta}}_{\sigma} &= \{\langle v_1, v_2\rangle \mid v_1 \in \mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma} \wedge v_2 \in \mathcal{V}_\nu[\![B]\!]^{\overline{\eta}}_{\sigma}\},\\
\mathcal{V}_\nu[\![A + B]\!]^{\overline{\eta}}_{\sigma} &= \{\mathsf{in}_1\, v \mid v \in \mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}\} \cup \{\mathsf{in}_2\, v \mid v \in \mathcal{V}_\nu[\![B]\!]^{\overline{\eta}}_{\sigma}\},\\
\mathcal{V}_\nu[\![A \to B]\!]^{\overline{\eta}}_{\sigma} &= \left\{\lambda x.t \,\middle|\, \forall \nu' \le \nu, \sigma' \sqsupseteq \mathrm{gc}(\sigma), \overline{\eta}' \sqsupseteq \overline{\eta}. \forall u \in \mathcal{V}_{\nu'}[\![A]\!]^{\overline{\eta}'}_{\sigma'}. t[u/x] \in \mathcal{T}_{\nu'}[\![B]\!]^{\overline{\eta}'}_{\sigma'}\right\},\\
\mathcal{V}_\nu[\![\Box A]\!]^{\overline{\eta}}_{\sigma} &= \{\mathsf{box}\, t \mid \forall \overline{\eta}'. t \in \mathcal{T}_\nu[\![A]\!]^{\overline{\eta}'}_{\emptyset}\},\\
\mathcal{V}_0[\![\bigcirc A]\!]^{\overline{\eta}}_{\sigma} &= \{l \mid l \in \mathsf{Loc}\}\\
\mathcal{V}_{\nu+1}[\![\bigcirc A]\!]^{\eta;\overline{\eta}}_{\sigma} &= \{l \mid \sigma(l) \in \mathcal{T}_\nu[\![A]\!]^{\overline{\eta}}_{\mathrm{gc}(\sigma)\checkmark\eta}\},\\
\mathcal{V}_\nu[\![\mathsf{Fix}\, \alpha.A]\!]^{\overline{\eta}}_{\sigma} &= \left\{\mathsf{into}(v) \,\middle|\, v \in \mathcal{V}_\nu[\![A[\bigcirc(\mathsf{Fix}\, \alpha.A)/\alpha]]\!]^{\overline{\eta}}_{\sigma}\right\}\\
\mathcal{T}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma} &= \left\{t \,\middle|\, \forall \sigma' \sqsupseteq_{\checkmark} \sigma. \exists \sigma'', v. \langle t; \sigma'\rangle \Downarrow \langle v; \sigma''\rangle \wedge v \in \mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma''}\right\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_\nu[\![\cdot]\!]^{\overline{\eta}}_{\sigma} &= \{\star\}\\
\mathcal{C}_\nu[\![\Gamma, x : A]\!]^{\overline{\eta}}_{\sigma} &= \left\{\gamma[x \mapsto v] \,\middle|\, \gamma \in \mathcal{C}_\nu[\![\Gamma]\!]^{\overline{\eta}}_{\sigma}, v \in \mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}\right\}\\
\mathcal{C}_\nu[\![\Gamma, \checkmark]\!]^{\overline{\eta}}_{\eta_N\checkmark\eta_L} &= \mathcal{C}_{\nu+1}[\![\Gamma]\!]^{\eta_L;\overline{\eta}}_{\eta_N}
\end{aligned}
$$

Garbage Collection:

$$
\begin{aligned}
\mathrm{gc}(\eta_L) &= \eta_L\\
\mathrm{gc}(\eta_N\checkmark\eta_L) &= \eta_L
\end{aligned}
$$

Fig. 8. Logical relation.

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 $\mathcal{C}_\nu[\![\Gamma]\!]^{\overline{\eta}}_{\sigma}$, which is given in Figure 8.

Theorem 5.1 (Fundamental Property). *Given $\Gamma \vdash t : A$, and $\gamma \in \mathcal{C}_\nu[\![\Gamma]\!]^{\overline{\eta}}_{\sigma}$, then $t\gamma \in \mathcal{T}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}$*

The proof of the fundamental property is a lengthy but entirely standard induction on the typing relation $\Gamma \vdash t : A$. Both Theorem 4.1 and Theorem 4.2 are then proved using the above theorem.

6 EMBEDDING RATTUS IN HASKELL

Our goal with Rattus is to combine the benefits of modal FRP with the practical benefits of FRP libraries. Because of the Fitch-style typing rules we cannot implement Rattus as a straightforward library of combinators. Instead we rely on a combination of a very simply library that implements the primitives of the language and a compiler plugin that performs some additional checks. We start with a description of the implementation followed by an illustration how the implementation is used in practice.

6.1 Implementation of Rattus

At its core, our implementation is consists of a very simple library that implements the primitives of our language (delay, adv, box, and unbox) so that they can be readily used in Haskell code. The

```
data ◯a = Delay a

delay :: a → ◯a
delay x = Delay x

adv :: ◯a → a
adv (Delay x) = x
```

```
data □a = Box a

box :: a → □a
box x = Box x

unbox :: □a → a
unbox (Box d) = d
```

Fig. 9. Implementation of Rattus primitives.

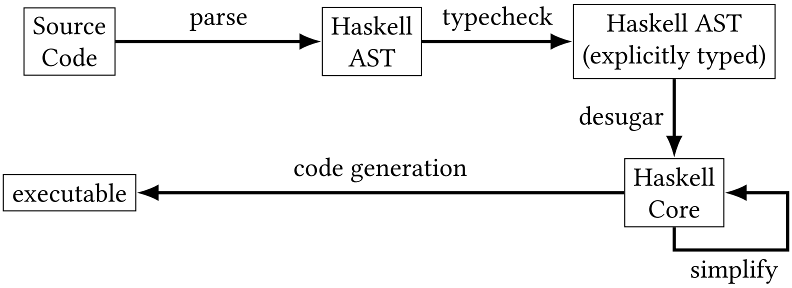

Fig. 10. Compiler phases of GHC (simplified).

library is given in its entirety (except for the *Stable* type class) in Figure 9. Both ◯ and □ are simple wrapper types, each with their own wrap and unwrap function. The constructors *Delay* and *Box* are not exported by the library, i.e. ◯ and □ 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: The 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 when Rattus's primitives may be used. So type checking a Rattus program boils down to first typechecking 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 use of delay, adv and box, but also due to guarded recursion. Similarly, we must make sure that delay and guarded recursive calls are only used in contexts where ✓ is absent, and adv is only used when a ✓ 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 10. There are two phases that are interesting for our implementation: the typechecking phase and the simplification phase. Simplification applies a series of transformations on the desugared abstract syntax tree (AST). This desugared language of GHC is called *Core* and GHC allows a plugin developer to add an additional transformation step by providing a suitable function of type *CoreProgram* → *CoreM CoreProgram*. Our goal is not to transform the Core AST but rather to perform an additional scope check on it. So our plugin implements a function

```
scopeCheck :: CoreProgram → CoreM CoreProgram
```

```
{-# 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. 11. Complete Rattus program.

that performs the requisite checks on the Core AST and if successful returns it with some modifications (see below). Otherwise, it uses the *CoreM* monad to print a helpful type error message. In general, one should avoid performing type-checking on a desugared representation as this results in poor error messages. However, in this case we only check for variable scopes so we are still able to give good 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 ◯ or function types in τ are nested under a □. However, we also want to support polymorphic types with type constraints such as in the *const* combinator:

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

The *Stable* type class is another primitive that is provided by our library and is defined as follows:

```
class StableInternal a where
class StableInternal a ⇒ Stable a where
```

We only export the *Stable* type class but not *StableInternal* to make sure the user of the language cannot implement the type class *Stable* for arbitrary types of their choosing. Our library does not implement instances of the *Stable* class either. Instead, such instances are derived by a second plugin that uses GHC's typechecker plugin API, which allows us to provide limited customisation to the type checking phase (see Figure 10). 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 our plugin, which will resolve it by performing the abovementioned syntactic check on τ.

The final component of our implementation is to make sure that it faithfully follows the operational semantics that we described for the core calculus in section 4.2. In particular, Rattus has a call-by-value semantics, i.e. arguments are evaluated before they are passed on to a function (except for delay and box). To this end, our implementation transforms all function applications so that arguments are evaluated to weak head normal form. This transformation is performed in the abovementioned *scopeCheck* function that is applied in GHC's simplification phase. If the Core AST satisfies Rattus's scoping rules then the AST is transformed in this way.

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 11 shows a complete program that illustrates the interaction between Haskell and Rattus. The language is imported via the *Rattus* module, with the *Rattus.Stream* providing a stream library (of which we have seen an excerpt in Figure 1). We only have 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:

$$\textbf{data}\ Trans\ a\ b = Trans\ (a \rightarrow (b, Trans\ a\ b))$$

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

The central ideas of functional reactive programming 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.

7.1 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 Bahr et al. [2019]; Cave et al. [2014]; Krishnaswami [2013]; Krishnaswami and Benton [2011]; Krishnaswami et al. [2012] are the only works to our knowledge giving operational guarantees. The work of Cave et al. [2014] shows how one can encode notions of fairness in modal FRP, if one replaces the guarded fixed point operator with more standard (co)recursion for (co)inductive types.

The guarded recursive types and 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] 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

$$\frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathsf{fix}\ x.t : \Box A} \qquad \frac{\Gamma \vdash t : \Box A \qquad \mathsf{token\text{-}free}(\Gamma')}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\ t : A} \qquad \frac{\Gamma, x : A, \Gamma' \vdash \qquad \mathsf{token\text{-}free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Fig. 12. Selected typing rules from Bahr et al. [2019].

modality in guarded recursion, to allow for a more direct style of programming for programs with complex input-output dependencies. Combining these ideas with the garbage collection results of this paper, however, seems very difficult.

7.2 Space leaks

The work by Krishnaswami [2013] and Bahr et al. [2019] 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. Moreover, he also implemented a compiler for his language, which translates FRP programs into JavaScript.

Like the present work, the Simply RaTT calculus of Bahr et al. uses a Fitch-style type system, which provides lighter syntax to interact with the □ and ◯ modality compared to Krishnaswami's use of qualifiers in his calculus. The latter is closely related to dual context systems and requires the use of pattern matching as elimination forms of the modalities (as opposed to the eliminators unbox and adv).

On the other hand Simply RaTT has a somewhat more complicated typing rule for guarded fixed points (cf. Figure 12). It uses a token ♯ (in addition to ✓) to serve the role that stabilisation of a context Γ to $\Gamma^{\Box}$ serves in Rattus. Moreover, fixed points produce terms of type $\Box A$ rather than just A. Taken together, this makes the syntax for guarded recursive function definitions more complicated. For example, the *map* function would be defined like this:

$$\begin{aligned}&\mathit{map} : \Box(a \to b) \to \Box(\mathit{Str}\ a \to \mathit{Str}\ b)\\ &\mathit{map}\ f\ \#\ (a :: \mathit{as}) = \mathsf{unbox}\ f\ \ a :: \mathit{map}\ f \circledast \mathit{as}\end{aligned}$$

Here, the ♯ is used to indicate that the argument f is to the left of the ♯ token and only because of the presence of this token we can use the unbox combinator on f (cf. Figure 12). Additionally, the typing of recursive definitions is somewhat awkward: *map* has return type $\Box(\mathit{Str}\ a \to \mathit{Str}\ b)$ but when used in a recursive call as seen above *map f* is of type $\bigcirc(\mathit{Str}\ a \to \mathit{Str}\ b)$ 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 #.

We argue that our typing system and syntax is simpler than both the work of Krishnaswami [2013] and Bahr et al. [2019], combining the simpler syntax of fixed points with the more streamlined syntax afforded by Fitch-style typing. In addition, our more general typing rule for variables (cf. Figure 12) also avoids the use of explicit operations for transporting stable variables over tokens, e.g. the *promote* operation that appears in both Krishnaswami [2013] and Bahr et al. [2019].

We should note that that Simply RaTT will reject some programs with time leaks, e.g. *leakyNats*, *leakySums2*, and *leakySums3* from section 4.4. We can easily write programs that are equivalent to *leakyNats* and *leakySums2*, that are well-typed Simply RaTT using tupling (essentially defining these functions simultaneously with *map*). On the other hand *leakySums3* cannot be expressed in Simply RaTT, essentially because the calculus does not support nested □ types. But a similar restriction can be implemented for Rattus, and indeed our implementation of Rattus will issue a warning when box or guarded recursion are nested.

8 DISCUSSION AND FUTURE WORK

We have shown that modal FRP 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. While these improvements in the underlying core calculus may appear mild, maintaining the operational properties along the way is a subtle balancing act.

This paper opens up many avenues for future work both on the implementation side and the underlying theory. We chose Haskell as our implementation 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 □ modality instead (cf. *const* vs. *constBox*).

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 [Bahr et al. 2017; Birkedal et al. 2018; Borghuis 1994; Clouston 2018] 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.

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 good 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 in Yampa signal functions are defined as a GADT that can handle some special cases much more efficiently.

REFERENCES

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

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 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.* IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/LICS.2017.8005097

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–27.

Andrew Barber. 1996. *Dual intuitionistic linear logic.* Technical Report. University of Edinburgh, Edinburgh, UK.

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. *arXiv:1804.05236 [cs]* (April 2018). http://arxiv.org/abs/1804.05236 00000 arXiv: 1804.05236.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *In Proc. of LICS.* IEEE Computer Society, Washington, DC, USA, 55–64. https://doi.org/10.2168/LMCS-8(4:1)2012

Valentijn Anton Johan Borghuis. 1994. *Coming to terms with modal logic: on the interpretation of modalities in typed lambda-calculus.* PhD Thesis. Technische Universiteit Eindhoven. http://repository.tue.nl/427575 00034.

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

Vikraman Chaudhury and Neel Krishnaswami. 2020. Recovering Purity with Comonads and Capabilities. (2020). ICFP 2020, to appear.

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, Springer International Publishing, Cham, 258–275.

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

Frederic Benton Fitch. 1952. *Symbolic logic, an introduction.* Ronald Press Co., New York, NY, USA.

Adrien Guatto. 2018. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 482–491.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2004. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming (Lecture Notes in Computer Science, Vol. 2638)*. Springer Berlin / Heidelberg. https://doi.org/10.1007/978-3-540-44833-4_6

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, Koen Claessen and Nikhil Swamy (Eds.). ACM, Philadelphia, PA, USA, 49–60. https://doi.org/10.1145/2103776.2103783

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

Wolfgang Jeltsch. 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* (Rome, Italy) *(PLPV '13)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

G. A. Kavvos. 2019. Modalities, Cohesion, and Information Flow. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 20:1–20:29. https://doi.org/10.1145/3290333 00000.

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, Boston, Massachusetts, USA, 221–232. https://doi.org/10.1145/2500365.2500588

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

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, PA, USA, 45–58. https://doi.org/10.1145/2103656.2103665

Bassel Mannaa and Rasmus Ejlers Møgelberg. 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, 23:1–23:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.23

Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. IEEE Computer Society, Washington, DC, USA, 255–266. https://doi.org/10.1109/LICS.2000.855774

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

Ross Paterson. 2001. A new notation for arrows. *ACM SIGPLAN Notices* 36, 10 (Oct. 2001), 229–240. https://doi.org/10.1145/507669.507664 00234.

Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. Association for Computing Machinery, Vancouver, BC, Canada, 302–314. https://doi.org/10.1145/2784731.2784752 00019.

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