JFP, **??** pages, 2018. © Cambridge University Press 2018 1
doi:10.1017/xxxxx

Modal FRP For All: Functional Reactive Programming Without Space Leaks in Haskell

PATRICK BAHR
IT University of Copenhagen (e-mail: paba@itu.dk)

Abstract

Functional reactive programming (FRP) provides a high-level interface for implementing reactive systems in a declarative manner. However, this high-level interface has to be carefully reigned in to ensure that programs can in fact be executed in practice. Specifically, one must ensure that FRP programs are 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 extends and simplifies previous modal FRP calculi while still maintaining the operational guarantees for productivity, causality, and space leaks. The simplified type system makes Rattus a practical programming language that can be integrated with existing functional programming languages. To demonstrate this, we have implemented a shallow embedding of Rattus in Haskell that allows the programmer to write Rattus code in familiar Haskell syntax and seamlessly integrate it with regular Haskell code. Thus Rattus combines the benefits enjoyed by FRP libraries such as Yampa, namely access to a rich library ecosystem (e.g. for graphics programming), with the strong operational guarantees offered by a bespoke type system.

To establish the productivity, causality, and memory properties of the language, we prove type soundness using a logical relations argument fully mechanised in the Coq proof assistant.

1 Introduction

Reactive systems perform an ongoing interaction with their environment, receiving inputs from the outside, changing their internal state and producing 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 challenging: The components of the reactive system are put together via a complex and often confusing web of callbacks and shared mutable state. As a consequence, individual components cannot be easily understood in isolation, which makes building and maintaining reactive systems in this manner difficult and error-prone.

Functional reactive programming (FRP), introduced by Elliott & Hudak (1997), tries to remedy this problem by introducing time-varying values (called *behaviours* or *signals*) and *events* as a means of communication between components in a reactive system instead of shared mutable state and callbacks. Crucially, signals and events are first-class values in FRP and can be freely combined and manipulated. These high-level abstractions not only provide a rich and expressive programming model. They also make it possible for us to reason about FRP programs by simple equational methods.

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

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

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

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

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

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

More recently, an alternative to this library approach has been developed (Jeffrey, 2014; Krishnaswami & Benton, 2011; Krishnaswami *et al.*, 2012; Krishnaswami, 2013; Jeltsch, 2013; Bahr *et al.*, 2019, 2021) 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:

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

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

That is, a stream of type *Str a* is an element of type *a* now and a stream of type *Str a* later, thus separating consecutive elements of the stream by one time step. Combining this modal type with guarded recursion (Nakano, 2000) in the form of a fixed point operator of type $(\bigcirc a \to a) \to a$ 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 is typically quite difficult to reason about their space and time cost. A reactive program that exhibits a gradually slower response time, i.e. its computations take longer and longer as time progresses, is said to have a *time leak*. Similarly, we say that a reactive program has a *space leak*, if its memory use is gradually increasing as time progresses, e.g. if it holds on to memory while continually allocating more.

Within both lines of work – the library approach and the modal types approach – there has been an effort to devise FRP languages that avoid *implicit* space leaks, 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 & 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.

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

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

Overview of Paper. Section 2 gives an overview of the Rattus language introducing the main concepts and their intuitions. Section 3 presents a case study of a simple FRP library based on streams and events, as well as an 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. We then reflect on these results and discuss the language design of Rattus. Section 5 gives an overview of the proof of our metatheoretical results. Section 6 describes how Rattus has been implemented as an embedded language in Haskell. Section 7 reviews related work and Section 8 discusses future work.

2 Introduction to Rattus

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

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

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

$\mathit{constInt} :: \mathit{Int} \to \mathit{Str}\ \mathit{Int}$
$\mathit{constInt}\ x = x ::: \mathsf{delay}\ (\mathit{constInt}\ x)$

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 $a \to \bigcirc a$, 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 rule 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 (Clouston, 2018): 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 several occurrences of ✓. 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 2.1.

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

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

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

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

```
leakyConst :: a → Str a
leakyConst x = x ::: delay (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. We can see that *leakyConst* would indeed cause a space leak by instantiating it to the type *leakyConst* :: *Str Int* → *Str* (*Str Int*): At each time step n it would have to store all previously observed input values from time step 0 to n, thus making its memory usage grow linearly with time.

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

```
loop :: Str Int
loop = loop    -- unguarded recursive call to loop is not allowed
```

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

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

```
inc :: Str Int → Str Int
inc (x ::: xs) = (x + 1) ::: delay (inc (adv xs))
```

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

$$\begin{array}{l} \mathit{tomorrow} :: \mathit{Str}\ \mathit{Int} \to \mathit{Str}\ \mathit{Int} \\ \mathit{tomorrow}\ (x ::: \mathit{xs}) = \mathsf{adv}\ \mathit{xs} \quad \text{-- adv is not allowed here} \end{array}$$

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 ✓:

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

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.

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

2.1 Stable types

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

However, 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 (even if a and b were stable) 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:

$$\begin{array}{l} \mathit{map} :: \Box(a \to b) \to \mathit{Str}\ a \to \mathit{Str}\ b \\ \mathit{map}\, f\ (x ::: \mathit{xs}) = \mathsf{unbox}\, f\ x ::: \mathsf{delay}\ (\mathit{map}\, f\ (\mathsf{adv}\ \mathit{xs})) \end{array}$$

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

The corresponding introduction form for $\Box$ does come with some restrictions. It has to make sure that boxed values do not refer to non-stable variables:

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

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

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

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

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

$$inc :: Str\ Int \to Str\ Int$$
$$inc = map\ (\mathsf{box}\ (+1))$$

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

$$constBox :: \Box a \to Str\ a$$
$$constBox\ a = \mathsf{unbox}\ a ::: \mathsf{delay}\ (constBox\ a)$$

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

$$const :: Stable\ a \Rightarrow a \to Str\ a$$
$$const\ x = x ::: \mathsf{delay}\ (const\ x)$$

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

```
leakyMap :: (a → 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 $\checkmark$ or whether they are of a stable type. One can think of top-level definitions being implicitly boxed when they are defined and implicitly unboxed when they are used later on.

2.2 Operational Semantics

As we have seen in the examples above, the purpose of the type modalities $\bigcirc$ and $\Box$ is to ensure that Rattus programs are causal and productive. Furthermore, the typing rules also ensure that Rattus has no implicit space leaks. In simple terms, the latter 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.

This is the same memory property Krishnaswami (2013) and Bahr *et al.* (2019, 2021) established for their modal FRP calculi. To achieve this, these calculi restrict guarded recursive definitions to only "look ahead" at most one time step. In the Fitch-style calculi of Bahr *et al.* (2019, 2021) this can be seen in the restriction to allow at most one ✓ in the typing context. For the same reason these two calculi also disallow function definitions in the context of a ✓. As a consequence, terms like $\mathsf{delay}(\mathsf{delay}\,0)$ and $\mathsf{delay}(\lambda x.x)$ do not type check in the calculi of Bahr *et al.* (2019, 2021).

Rattus lifts these restrictions on ticks and instead refines the operational semantics of the language. At first glance one might think that allowing multiple ticks can be accommodated by extending the time one has to keep temporal values in memory accordingly, i.e. we may safely garbage collect temporal values after $n+1$ time steps, if we allow at most n ticks. However, this turns out to be neither enough nor necessary. On the one hand, even allowing just two ticks would require us to keep temporal values in memory indefinitely, i.e. it would permit implicit space leaks. On the other hand, if we change the evaluation strategy, we can still garbage collect all temporal values after two time steps, no matter how many ticks were involved in type checking the program.

Similarly to the abovementioned earlier calculi of Krishnaswami and Bahr *et al.*, Rattus uses an eager evaluation strategy except for delay and box. That is, arguments are evaluated to values before they are passed on to functions other than delay and box. Where Rattus differs is the evaluation of adv. Simply put, Rattus follows the *temporal* semantics of adv: Recall that $\mathsf{delay}\,t$ delays the computation t by one time step and that adv reverses such a delay. The operational semantics of Rattus reflects this intuition by first evaluating every term t that occurs as $\mathsf{delay}\,(\ldots \mathsf{adv}\,t \ldots)$ before evaluating delay. To enforce this evaluation strategy, Rattus programs are transformed before execution employing two rewrite rules (see Section 4.2.1).

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

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

Applying *stutter* to 0 would construct a stream of numbers $0, 0, 1, 1, 2, 2, \ldots$. In order to implement *stutter* in the more restrictive language of Krishnaswami (2013) and Bahr *et al.*

(2019, 2021) we would need to decompose it into two mutually recursive functions (Bahr *et al.*, 2021).

The operational semantics 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

In this section we showcase how Rattus can be used for reactive programming. To this end, we implement a small library of combinators for programming with streams and events. We then use this library to implement a simple game. Finally, we implement a Yampa-style library and re-implement the game using that library instead.

3.1 Programming with streams and events

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

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

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

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

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

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 in response to the occurrence of an event. There are different ways to represent events. The simplest representation defines events of type a as streams of type *Maybe a*. However, we will use the strict variant of the *Maybe* type:

data $Maybe'\ a = Just'\ !\ a \mid Nothing'$

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. In this implementation, the behaviour changes *every time* an event occurs, not only the first time. For a one-shot variant of *switch*, we would just have to change the second equation to $switch\ \_\ (Just'\ as ::: \_) = as$.

```
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 Event a = Str (Maybe' a)

switch :: Str a → Event (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) → Event (Str a → Str b) → (Str a → Str b)
switchTrans f es as = switchTrans' (f as) es as

switchTrans' :: Str b → Event (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.

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'*, which takes just a stream as its first argument instead of a stream function.

3.2 A simple reactive program

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

```
pong :: Str Input → Str (Pos ⊗ Float)
pong inp = zip ball pad where
  pad :: Str Float
  pad = padPos inp
```

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

```
  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 compute the position and the velocity of the object, while consuming the input stream. The velocity is only needed to compute the position and is therefore projected away afterwards 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 (with velocity $(20,50)$).

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

To achieve this behaviour we use the *switchTrans* combinator, which we initialise with the original behaviour of the ball. The event that will trigger the switch is constructed by mapping *ballTrig* over the input stream, which will create an event of type $Event\ (Str\ (Float \otimes Input) \to Str\ Pos)$, 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 in a way that guarantees 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. By implementing an arrowized FRP library in Rattus instead of plain Haskell, we can not only guarantee causality but also productivity and the absence of implicit space leaks. Furthermore, this demonstrates that Rattus can also be used to implement a continuous-time FRP library, in contrast to the FRP library from Section 3.1, which is discrete-time.

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 $ballPos :: SF\ (Float \otimes Input)\ Pos$ and $padPos :: SF\ Input\ Float$ describing the positions of the ball and the paddle from our game in Section 3.2, we can combine these as follows:

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

The Rattus definition of *SF* is almost identical to the original Haskell definition from Nilsson *et al.* (2002). We only 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 with the exception of the *arr* method. In fact, we cannot implement *arr* in Rattus at all. Because the first argument is not stable, it falls out of scope in the recursive call:

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

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

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

In other words, in conventional arrowized FRP, the *arr* method is a potential source for space leaks. To avoid such potential space leaks, we have to replace *arr* with the more restrictive variant *arrBox*. But fortunately, this does not prevent us from using the arrow notation: Rattus treats *arr f* as a short hand for *arrBox* (box *f*), which allows us to use the arrow notation while making sure that box *f* is well-typed, i.e. *f* only refers to variables of stable type.

The *Arrow* type class only provides a basic interface for constructing *static* signal functions. To permit dynamic behaviour we need to provide additional combinators, e.g. 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
```

```
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.

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 can be endowed with a very general recursion principle by instantiating the *loop* method in the *ArrowLoop* type class shown in Figure 2. However, *loop* cannot be implemented in Rattus as it would break the productivity property. Instead, we implement a more restricted recursion principle that corresponds to guarded recursion:

```
loopPre :: d → SF (b ⊗ d) (c ⊗ ◯d) → SF b c
```

Intuitively speaking, this combinator constructs a signal function from b to c with the help of an internal state of type d. The first argument initialises the state, and the second argument is a signal function that turns input of type b into output of type c while also updating the internal state. Apart from the addition of the $\bigcirc$ modality and strict pair types, this definition has the same type as Yampa's *loopPre*.

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 (calcNewVelocity 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
```

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

4 Core Calculus

In this section we present the core calculus of Rattus, which we call $\lambda_{\checkmark\!\!\checkmark}$. The purpose of $\lambda_{\checkmark\!\!\checkmark}$ 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: a simply typed lambda calculus extended with guarded recursive types $\mathsf{Fix}\,\alpha.A$, a guarded fixed point combinator, 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 $\lambda_{\checkmark\!\!\checkmark}$. Besides guarded recursive types, guarded fixed points, 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 is represented 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 $\checkmark$ tokens. If a typing context contains no $\checkmark$, we call it *tick-free*. The complete set of typing rules for $\lambda_{\checkmark\!\!\checkmark}$ is given in Figure 5. The typing rules for Rattus presented in Section 2 appear in the same form also here, except for replacing Haskell's :: operator with the more standard notation. The remaining typing rules are entirely standard, except for the typing rule for the guarded fixed point combinator fix.

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

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

$$\begin{array}{l} \mathit{constInt} :: \mathit{Int} \to \mathit{Str}\ \mathit{Int} \\ \mathit{constInt}\ x = x ::: \mathsf{delay}\ (\mathit{constInt}\ x) \end{array}$$

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

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

where the stream cons operator $s ::: t$ is shorthand for $\mathsf{into}\ \langle s, t\rangle$. The variable r is of type $\Box(\bigcirc(\mathit{Int} \to \mathit{Str}\ \mathit{Int}))$ and applying unbox followed by adv turns it into type $\mathit{Int} \to \mathit{Str}\ \mathit{Int}$. Moreover, the restriction that recursive calls must occur in a context with $\checkmark$ makes sure

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

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

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

Fig. 4. Well-formed contexts

that this transformation from recursion notation to fixed point combinator results in a well-typed $\lambda_{\checkmark\!\!\checkmark}$ term.

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

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

$$\begin{aligned}
&\mathit{leakyMap} :: (a \to b) \to \mathit{Str}\ a \to \mathit{Str}\ b\\
&\mathit{leakyMap}\, f = \mathit{run}\\
&\quad \textbf{where}\ \mathit{run} :: \mathit{Str}\ a \to \mathit{Str}\ b\\
&\qquad\quad \mathit{run}\ (x ::: xs) = f\ x ::: \mathsf{delay}\ (\mathit{leakyMap}\ (\mathsf{adv}\ xs))
\end{aligned}$$

Translated into $\lambda_{\checkmark\!\!\checkmark}$, the definition looks like this:

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

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.

4.2 Operational Semantics

The operational semantics is given in two steps: To execute a program of $\lambda_{\checkmark\!\!\checkmark}$ it is first translated into a more restrictive variant of $\lambda_{\checkmark\!\!\checkmark}$, which we call $\lambda_{\checkmark}$. The resulting $\lambda_{\checkmark}$ program is then executed using an abstract machine that ensures the absence of implicit space leaks by construction.

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

$$\frac{\Gamma \vdash_{\checkmark\!\checkmark} s : \mathsf{Int} \qquad \Gamma \vdash_{\checkmark\!\checkmark} t : \mathsf{Int}}{\Gamma \vdash_{\checkmark\!\checkmark} s + t : \mathsf{Int}} \qquad \frac{\Gamma, x : A \vdash_{\checkmark\!\checkmark} t : B}{\Gamma \vdash_{\checkmark\!\checkmark} \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash_{\checkmark\!\checkmark} s : A \qquad \Gamma, x : A \vdash_{\checkmark\!\checkmark} t : B}{\Gamma \vdash_{\checkmark\!\checkmark} \mathsf{let}\, x = s \,\mathsf{in}\, t : B}$$

$$\frac{\Gamma \vdash_{\checkmark\!\checkmark} t : A \to B \qquad \Gamma \vdash_{\checkmark\!\checkmark} t' : A}{\Gamma \vdash_{\checkmark\!\checkmark} t\, t' : B} \qquad \frac{\Gamma \vdash_{\checkmark\!\checkmark} t : A \qquad \Gamma \vdash_{\checkmark\!\checkmark} t' : B}{\Gamma \vdash_{\checkmark\!\checkmark} \langle t, t' \rangle : A \times B}$$

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

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

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

$$\frac{\Gamma \vdash_{\checkmark\!\checkmark} t : A[\bigcirc(\mathsf{Fix}\, \alpha.A)/\alpha]}{\Gamma \vdash_{\checkmark\!\checkmark} \mathsf{into}\, t : \mathsf{Fix}\, \alpha.A} \qquad \frac{\Gamma \vdash_{\checkmark\!\checkmark} t : \mathsf{Fix}\, \alpha.A}{\Gamma \vdash_{\checkmark\!\checkmark} \mathsf{out}\, t : A[\bigcirc(\mathsf{Fix}\, \alpha.A)/\alpha]}$$

$$\frac{\Gamma^{\Box}, x : \Box(\bigcirc A) \vdash_{\checkmark\!\checkmark} t : A}{\Gamma \vdash_{\checkmark\!\checkmark} \mathsf{fix}\, x.t : A}$$

Fig. 5. Typing rules.

4.2.1 *Translation to $\lambda_{\checkmark}$*

The $\lambda_{\checkmark}$ calculus restricts typing contexts to contain at most one $\checkmark$ and restricts the typing rules for lambda abstraction and delay as follows:

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

The construction $|\Gamma|$ turns Γ into a tick-free context, which ensures that we have at most one $\checkmark$ – even for nested occurrences of delay – and that lambda abstractions are not in the scope of a $\checkmark$. All other typing rules are the same as for $\lambda_{\checkmark\!\checkmark}$.

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

$$\begin{array}{rl} \mathsf{delay}(C[\mathsf{adv}\, t]) \longrightarrow \mathsf{let}\, x = t \,\mathsf{in}\, \mathsf{delay}(C[\mathsf{adv}\, x]) & \text{if } t \text{ is not a variable} \\ \lambda x.C[\mathsf{adv}\, t] \longrightarrow \mathsf{let}\, y = \mathsf{adv}\, t \,\mathsf{in}\, \lambda x.(C[y]) & \end{array}$$

where C is a term with a single hole that does not occur in the scope of delay, adv, box, fix, or lambda abstraction.

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

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

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

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

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

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

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

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

Exhaustiveness. Finally, we need to show that $\vdash_{\checkmark\checkmark} t : A$ with $t \not\longrightarrow$ implies $\vdash_{\checkmark} t : A$, i.e. if we cannot rewrite t any further it must be typable in $\lambda_{\checkmark}$ as well. In order to prove this property by induction we must generalise it to open terms: If $\Gamma \vdash_{\checkmark\checkmark} t : A$ for a context Γ with at most one tick and $t \not\longrightarrow$, then $\Gamma \vdash_{\checkmark} t : A$. We prove this implication by induction on t and a case distinction on the last typing rule in the derivation of $\Gamma \vdash_{\checkmark\checkmark} t : A$. For almost all cases, $\Gamma \vdash_{\checkmark} t : A$ follows from the induction hypothesis since we find a corresponding typing rule in $\lambda_{\checkmark}$ that is either the same as in $\lambda_{\checkmark\checkmark}$ or has a side condition is satisfied by our assumption that Γ have at most one tick. We are thus left with two interesting cases: the typing rules for delay and lambda abstraction, given that Γ contains exactly one tick (the zero-tick cases are trivial). Each of these two cases follows from the induction hypothesis and one of the following two properties:

(1) If $\Gamma_1, \checkmark, \Gamma_2 \vdash_{\checkmark\!\!\!\checkmark} t : A$, Γ_2 contains a tick, and delay $t \not\longrightarrow$, then $\Gamma_1^{\square}, \Gamma_2 \vdash_{\checkmark\!\!\!\checkmark} t : A$.
(2) If $\Gamma_1, \checkmark, \Gamma_2 \vdash_{\checkmark} t : A$ and $\lambda x.t \not\longrightarrow$, then $\Gamma_1^{\square}, \Gamma_2 \vdash_{\checkmark} t : A$.

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

4.2.2 Abstract machine for $\lambda_{\checkmark}$

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

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

The machine consists of two components: the *evaluation semantics*, presented in Figure 6, which describes the operational behaviour of $\lambda_{\checkmark}$ 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 $l \mapsto t$.

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

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

$$\frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \mathsf{in}_i(t);\sigma\rangle \Downarrow \langle \mathsf{in}_i(v);\sigma'\rangle} \qquad \frac{\langle s;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \langle t[v/x];\sigma'\rangle \Downarrow \langle v';\sigma''\rangle}{\langle \mathsf{let}\ x = s\ \mathsf{in}\ t;\sigma\rangle \Downarrow \langle v';\sigma''\rangle}$$

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

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

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

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

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \mathsf{into}\,v;\sigma'\rangle}{\langle \mathsf{out}\,t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle} \qquad \frac{\langle t[\mathsf{box}\,(\mathsf{delay}\,(\mathsf{fix}\,x.t))/x];\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{fix}\,x.t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

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.

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

In principle, the abstract machine could also be used for $\lambda_{\checkmark\!\!\checkmark}$ terms directly, without transforming them to $\lambda_{\checkmark}$ first. However, we can construct a well-typed $\lambda_{\checkmark\!\!\checkmark}$ term for which the machine will try to dereference a heap location that has previously been garbage collected. In other words, programs in $\lambda_{\checkmark\!\!\checkmark}$ may exhibit implicit space leaks, if we just run them unaltered on the abstract machine.

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

$$\mathsf{fix}\ r.\lambda x.\mathsf{delay}(\mathsf{head}(\mathsf{adv}\, x) ::: \mathsf{adv}(\mathsf{unbox}\, r)(\mathsf{delay}(\mathsf{adv}(\mathsf{tail}(\mathsf{adv}\, x)))))$$

Here the abstract machine would get stuck trying to dereference a heap location that was previously garbage collected. The problem is that the second occurrence of x is nested below two occurrences of delay, which means when $\mathsf{adv}\, x$ is evaluated, the heap location bound to x is two time steps old and has been garbage collected already. Importantly, this problem would occur even if we increased the number of heaps from two to any other fixed upper bound. We would need an unbounded number of heaps to safely execute this term.

On the other hand, if we apply the rewrite rules from Section 4.2.1, we obtain the following $\lambda_{\checkmark}$ term that is safe to execute on the abstract machine and thus does not suffer from space leaks:

$$\begin{aligned}&\mathsf{fix}\ r.\lambda x.\mathsf{let}\ r' = \mathsf{unbox}\ r\\ &\qquad \mathsf{in}\ \mathsf{delay}(\mathsf{head}(\mathsf{adv}\, x) ::: \mathsf{adv}\, r'(\mathsf{let}\ y = \mathsf{tail}(\mathsf{adv}\, x)\ \mathsf{in}\ \mathsf{delay}(\mathsf{adv}\, y)))\end{aligned}$$

In the next section, we present our main theoretical result, namely that all closed $\lambda_{\checkmark}$ terms of the appropriate type are safe to execute on the abstract machine.

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 $Str\, A \to Str\, B$. The former just produce infinite streams of values of type A, whereas the latter are reactive processes that produce a value of type B for each input value of type A.

4.3.1 Productivity of the step semantics

The step semantics $\overset{v}{\Longrightarrow}$ from Figure 7 describes the unfolding of streams of type $Str\, A$. Given a closed term $\vdash_{\checkmark} t : \mathsf{Str}\, A$, 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 step semantics, we evaluate $\langle t_i; \eta_i\checkmark\rangle \Downarrow \langle v_i ::: l; \eta_i' \checkmark \eta_{i+1}\rangle$, where η_i' is η_i but possibly extended with some additional delayed computations and η_{i+1} is the new heap with delayed computations for the next time step. Crucially, the old heap η_i' is thrown away. That is, by construction, old data is not implicitly retained but garbage collected immediately after we completed the current time step.

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

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

This definition translates to the $\lambda_{\not\checkmark}$ term $\mathsf{fix}\ r.\lambda n.n ::: \mathsf{delay}(\mathsf{adv}\ (\mathsf{unbox}\ r)\ (n+\overline{1}))$, which, in turn, rewrites into the following $\lambda_{\checkmark}$ term:

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

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

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

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

Our main result is that the execution of $\lambda_{\checkmark}$ terms by the machine described in Figure 6 and 7 is safe. To describe the type of the produced values precisely, we need to restrict ourselves to streams over types whose evaluation is not suspended, which excludes function and modal types. This idea is expressed in the notion of *value types*, defined by the following grammar:

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

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

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

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

4.3.2 Causality of the step semantics

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

$$\langle t\ (\mathsf{adv}\ l^*); \emptyset\rangle \stackrel{v_0/v_0'}{\Longrightarrow} \langle t_1; \eta_1\rangle \stackrel{v_1/v_1'}{\Longrightarrow} \langle t_2; \eta_2\rangle \stackrel{v_2/v_1'}{\Longrightarrow} \dots$$

where each output value v_i' has type B.

The definition of $\stackrel{v/v'}{\Longrightarrow}$ 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. As we can see above, this stand-in value l^* is passed on to the stream function in the form of the argument $\mathsf{adv}\, l^*$. Then, in each step, we evaluate the current term t_i in the current heap η_i

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

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 $\lambda_{\checkmark}$ calculus:

$$run = \mathsf{fix}\, r.\lambda acc.\lambda s.\mathsf{let}\, x = \mathsf{head}\, s \,\mathsf{in}\,\mathsf{let}\, xs = \mathsf{tail}\, s \,\mathsf{in}\,\mathsf{let}\, acc' = acc + x \,\mathsf{in}$$
$$\mathsf{let}\, r' = \mathsf{unbox}\, r \,\mathsf{in}\, acc' ::: \mathsf{delay}(\mathsf{adv}\, r'\, acc' (\mathsf{adv}\, xs))$$
$$sum = run\, \overline{0}$$

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

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

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 (2, 13, and 18, respectively) and the tail of the current input stream.

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

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

$$\langle t\ (\mathsf{adv}\, l^*); \emptyset\rangle \overset{v_0/v_0'}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_2/v_2'}{\Longrightarrow} \dots \qquad \text{such that } \vdash_{\checkmark} v_i' : B \text{ for all } i \geq 0.$$

Since the operational semantics is deterministic, in each step $\langle t_i; \eta_i \rangle \overset{v_i/v'_i}{\Longrightarrow} \langle t_{i+1}; \eta_{i+1} \rangle$ 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' core calculus, we should make sure that we understand what they mean in practice and what their limitations are. In simple terms, the productivity and causality properties established by Theorem 4.2 and Theorem 4.3 state that reactive programs in Rattus can be executed effectively – they always make progress and never depend on data that is not yet available. However, Rattus allows calling general Haskell functions, for which we can make no such operational guarantees. This trade-off is intentional as we wish to make Haskell's rich library ecosystem available to Rattus. Similar trade-offs are common in foreign function interfaces that allow function calls into another language. For instance Haskell code may call C functions.

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

Given a strict list type

$$\textbf{data}\ List\ a = Nil \mid\ !a :!\ !(List\ a)$$

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

$$buffer :: Stable\ a \Rightarrow Str\ a \to Str\ (List\ a)$$
$$buffer = scan\ (\mathsf{box}\ (\lambda xs\ x \to x :!\ xs))\ Nil$$

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

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

At each time step this function adds the current input integer to the buffer of type *List Int* and then computes the sum of the current value of that buffer. This function exhibits both a space leak (buffering a steadily growing list of numbers) and a time leak (the time to compute each element of the resulting stream increases at each step). However, these leaks are explicit in the program.

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

$$leakySums2 :: Str\ Int \to Str\ Int$$
$$leakySums2\ (x ::: xs) = x ::: \mathsf{delay}\ (map\ (\mathsf{box}\ (+x))\ (leakySums2\ (\mathsf{adv}\ xs)))$$

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

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

4.5 Language Design Considerations

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

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

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

$$map : \Box(a \to b) \to \Box(Str\ a \to Str\ b)$$
$$map\, f\ \#\ (a :: as) = \mathsf{unbox}\, f\ a :: (map\, f \circledast as)$$

Here, # is used to indicate that the argument f is to the left of the $\sharp$ token and only because of the presence of this token can we use the unbox combinator on f (cf. Figure 8).

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

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

Additionally, the typing of recursive definitions is somewhat confusing: *map* has return type $\Box(Str\, a \to Str\, b)$ but when used in a recursive call as seen above, *map f* is of type $\bigcirc(Str\, a \to Str\, b)$ instead. Moreover, we cannot call *map* recursively on its own. All recursive calls must be of the form *map f*, the exact pattern that appears to the left of the #. This last restriction rules out the following variant of *map* that is meant to take two functions and alternately apply them to a stream:

$alterMap : \Box(a \to b) \to \Box(a \to b) \to \Box(Str\, a \to Str\, b)$
$alterMap\, f\, g\, \#\, (a :: as) = \mathsf{unbox}\, f\, a :: (alterMap\, g\, f \circledast as)$

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

$alterMap : \Box(a \to b) \to \Box(a \to b) \to Str\, a \to Str\, b$
$alterMap\, f\, g\, (a :: as) = f\, a :: (\mathsf{delay}\, (alterMap\, g\, f) \circledast as)$

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

$pong' :: \Box(Str\, Input \to Str\, (Pos \otimes Float))$
$pong'\, \#\, inp = \mathsf{unbox}\, zip\, ball\, pad$ **where**
$\quad pad = \mathsf{unbox}\, padPos\, inp$
$\quad ball = \mathsf{unbox}\, switchTrans\, (\mathsf{unbox}\, ballPos\, inp)$
$\qquad\qquad (\mathsf{unbox}\, (triggerMap\, ballTrig)\, inp)$
$\qquad\qquad (\mathsf{unbox}\, zip\, pad\, inp)$

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

Conversely, if we keep the unbox combinator of Simply RaTT but lift some of the restrictions regarding the $\sharp$ token such as allowing ✓ not only to the right of a $\sharp$ or allowing more than one $\sharp$ token, then we would break the garbage collection property and thus permit leaky programs. In such a modified version of Simply RaTT, we would be able to type check the following term:

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

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

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

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

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

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

$$f : A \to \square B \vdash \lambda x.\mathsf{box}(\mathsf{unbox}(f\,x)) : A \to \square B$$

in Simply RaTT but not in Rattus. However, Rattus has a slightly different eta-expansion for this type instead:

$$f : A \to \square B \vdash_{\checkmark} \lambda x.\mathsf{let}\; x' = \mathsf{unbox}(f\,x)\;\mathsf{in}\;\mathsf{box}\,x' : A \to \square B$$

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

$$f : A \to \square B\;\;\mathsf{now} \vdash \lambda x.\mathsf{let}\;\mathsf{stable}(x') = f\,x\;\mathsf{in}\;\mathsf{stable}\,x' : A \to \square B$$

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

$$x : \square(\square A) \vdash_{\checkmark} \mathsf{box}(\mathsf{box}(\mathsf{unbox}(\mathsf{unbox}\,x))) : \square(\square A)$$
$$x : \bigcirc(\bigcirc A) \vdash_{\checkmark} \mathsf{delay}(\mathsf{delay}(\mathsf{adv}(\mathsf{adv}\,x))) : \bigcirc(\bigcirc A)$$
$$f : \bigcirc(A \to B) \vdash_{\checkmark} \mathsf{delay}(\lambda x.\mathsf{adv}\,f\,x) : \bigcirc(A \to B)$$

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

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

5 Meta Theory

Our goal is to show that Rattus's core calculus $\lambda_{\checkmark}$ enjoys the three central operational properties: productivity, causality, and absence of implicit space leaks. These properties are stated in Theorem 4.2 and Theorem 4.3, 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 here.

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 & 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 $l \in \mathsf{dom}\,(\eta)$. 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, whereas $\sqsubseteq_{\checkmark}$ is more general and permits introducing an arbitrary 'now' heap if none is present.

Given these orderings we define two logical relations, the value relation $\mathcal{V}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}$ and the term relation $\mathcal{T}_\nu[\![A]\!]^{\overline{\eta}}_{\sigma}$. Both are defined in Figure 9 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.

$$|\alpha| = |\bigcirc A| = |\mathsf{Int}| = |1| = 1$$
$$|A \times B| = |A + B| = |A \to B| = 1 + |A| + |B|$$
$$|\square A| = |\mathsf{Fix}\,\alpha.A| = 1 + |A|$$

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 $\mathsf{gc}(\sigma)\checkmark\eta$. Here $\mathsf{gc}(\sigma)$ denotes the garbage collection of the store σ as defined in Figure 9. It simply drops the 'now' heap if present. To see how this definition captures causality we have to look 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 $\mathsf{gc}(\sigma)\checkmark\eta$, whereas the old 'later' heap in σ becomes the new 'now' heap.

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

Theorem 5.1 (Fundamental Property). If $\Gamma \vdash_{\checkmark} t : A$, and $\gamma \in \mathscr{C}_\mathcal{V}[\![\Gamma]\!]^{\overline{\eta}}_{\sigma}$, then $t\gamma \in \mathscr{T}_\mathcal{V}[\![A]\!]^{\overline{\eta}}_{\sigma}$.

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

6 Embedding Rattus in Haskell

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

6.1 Implementation of Rattus

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

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

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

GARBAGE COLLECTION:

$$
\mathsf{gc}(\eta_L) = \eta_L \qquad \mathsf{gc}(\eta_N \checkmark \eta_L) = \eta_L
$$

Fig. 9. Logical relation.

```
data ◯a = Delay a        data □a = Box a         class StableInternal a where
delay :: a → ◯a          box :: a → □a           class StableInternal a ⇒ Stable a where
delay x = Delay x        box x = Box x
adv :: ◯a → a            unbox :: □a → a
adv (Delay x) = x        unbox (Box d) = d
```

Fig. 10. Implementation of Rattus primitives.

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

The design of this plugin follows the simple observation that any Rattus program is also a Haskell program but with more restrictive rules for variable scope and for where Rattus's primitives may be used. So type checking a Rattus program boils down to first 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. Additionally, we must make sure that both guarded recursive calls and adv only appear in a context where ✓ is present.

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

After type checking the Haskell abstract syntax tree (AST), GHC passes the resulting typed AST on to the scope checking component of the Rattus plugin, which checks the abovementioned stricter scoping rules. GHC then desugars the typed AST into the intermediate language *Core*. GHC then performs a number of transformation passes on this intermediate representation, the first two of these are provided by the Rattus plugin: First, we exhaustively apply the two rewrite rules from Section 4.2.1 to transform the program into a single-tick form according to the typing rules of $\lambda_{\checkmark}$. Then we transform the resulting code so that Rattus programs adhere to the call-by-value semantics. To this end, the plugin's *strictness* pass transforms all function applications so that arguments are evaluated to weak head normal form before they are passed to the function. In addition, this *strictness* pass also checks that Rattus code only uses strict data types and issues a warning if lazy data types are used, e.g. Haskell's standard list and pair types.

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

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

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

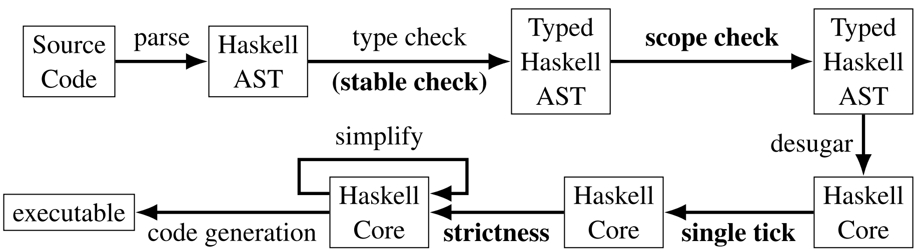

Fig. 11. Compiler phases of GHC (simplified) extended with Rattus plugin (highlighted in bold).

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

6.2 Using Rattus

To write Rattus code inside Haskell one must use GHC with the flag `-fplugin=Rattus.Plugin`, which enables the Rattus plugin described above. Figure 12 shows a complete program that illustrates the interaction between Haskell and Rattus. The language is imported via the *Rattus* module, with the *Rattus.Stream* module providing a stream library (of which we have seen an excerpt in Figure 1). The program contains only one Rattus function, *summing*, which is indicated by an annotation. This function uses the *scan* combinator to define a stream transducer that sums up its input stream. Finally, we use the *runTransducer* function that is provided by the *Rattus.ToHaskell* module. It turns a stream function of type $Str\ a \to Str\ b$ into a Haskell value of type $Trans\ a\ b$ defined as follows:

$$\mathbf{data}\ Trans\ a\ b = Trans\ (a \to (b, Trans\ a\ b))$$

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 & 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 & 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

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

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

Fig. 12. Complete Rattus program.

does not imply the absence of space leaks, but rather that in principle it can be implemented without space leaks.

Modal FRP calculi The idea of using modal type operators for reactive programming goes back to Jeffrey (2012), Krishnaswami & Benton (2011), and Jeltsch (2013). One of the inspirations for Jeffrey (2012) was to use linear temporal logic (Pnueli, 1977) as a programming language through the Curry-Howard isomorphism. The work of Jeffrey and Jeltsch has mostly been based on denotational semantics, and to our knowledge Krishnaswami & Benton (2011); Krishnaswami *et al.* (2012); Krishnaswami (2013); Cave *et al.* (2014); Bahr *et al.* (2019, 2021) are the only works giving operational guarantees. In addition, the calculi of Bahr *et al.* (2021) and Cave *et al.* (2014) can be used to encode liveness properties in modal FRP.

Guarded recursive types and the guarded fixed point combinator originate with Nakano (2000), but have since been used for constructing logics for reasoning about advanced programming languages (Birkedal *et al.*, 2011) using an abstract form of step-indexing (Appel & 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 & Møgelberg, 2018) reveals the difference from the more standard dual context approaches to modal logics, such as Dual Intuitionistic Linear Logic (Barber, 1996): In the latter, the modal operator is implicitly applied to the type of all variables in one context, in the Fitch-style, placing a tick in a context corresponds to applying a *left adjoint* to the modal operator to the context. Guatto (2018) introduced the notion of time warp and the warping modality, generalising the delay modality in guarded recursion, to allow for a more direct style of programming for programs with complex input-output dependencies.

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

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.

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

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

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

Bahr, P., Grathwohl, H. B. and Møgelberg, R. E. (2017) The clocks are ticking: No more delays! *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017* pp. 1–12. IEEE Computer Society.

Bahr, P., Graulund, C. U. and Møgelberg, R. E. (2019) Simply ratt: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages*

3(ICFP):1–27.

Bahr, P., Graulund, C. U. and Møgelberg, R. E. (2021) *Diamonds are not Forever: Liveness in Reactive Programming with Guarded Recursion*. POPL 2021, to appear.

Barber, A. (1996) *Dual intuitionistic linear logic*. Tech. rept. University of Edinburgh, Edinburgh, UK.

Birkedal, L., Møgelberg, R. E., Schwinghammer, J. and Støvring, K. (2011) First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. *In Proc. of LICS* pp. 55–64. IEEE Computer Society.

Birkedal, L., Clouston, R., Mannaa, B., Møgelberg, R. E., Pitts, A. M. and Spitters, B. (2018) Modal Dependent Type Theory and Dependent Right Adjoints. *arXiv:1804.05236 [cs]* Apr. 00000 arXiv: 1804.05236.

Borghuis, V. A. J. (1994) *Coming to terms with modal logic: on the interpretation of modalities in typed lambda-calculus*. PhD Thesis, Technische Universiteit Eindhoven. 00034.

Cave, A., Ferreira, F., Panangaden, P. and Pientka, B. (2014) Fair Reactive Programming. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14, pp. 361–372. ACM.

Chaudhury, V. and Krishnaswami, N. (2020) *Recovering Purity with Comonads and Capabilities*. ICFP 2020, to appear.

Clouston, R. (2018) Fitch-style modal lambda calculi. Baier, C. and Dal Lago, U. (eds), *Foundations of Software Science and Computation Structures*, vol. 10803, pp. 258–275. Cham: Springer International Publishing, for Springer.

Elliott, C. and Hudak, P. (1997) Functional reactive animation. *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97, pp. 263–273. ACM.

Fitch, F. B. (1952) *Symbolic logic, an introduction*. Ronald Press Co.

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

Hudak, P., Courtney, A., Nilsson, H. and Peterson, J. (2004) Arrows, Robots, and Functional Reactive Programming. *Advanced Functional Programming*. Lecture Notes in Computer Science 2638. Springer Berlin / Heidelberg.

Jeffrey, A. (2012) LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. Claessen, K. and Swamy, N. (eds), *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012* pp. 49–60. ACM.

Jeffrey, A. (2014) Functional reactive types. *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS '14, pp. 54:1–54:9. ACM.

Jeltsch, W. (2013) Temporal logic with "until", functional reactive programming with processes, and concrete process categories. *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*. PLPV '13, pp. 69–78. ACM.

Kavvos, G. A. (2019) Modalities, Cohesion, and Information Flow. *Proc. ACM Program. Lang.* **3**(POPL):20:1–20:29. 00000.

Krishnaswami, N. R. (2013) Higher-order Functional Reactive Programming Without Spacetime Leaks. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13, pp. 221–232. ACM.

Krishnaswami, N. R. and Benton, N. (2011) Ultrametric semantics of reactive programs. *2011 IEEE 26th Annual Symposium on Logic in Computer Science* pp. 257–266. IEEE Computer Society.

Krishnaswami, N. R., Benton, N. and Hoffmann, J. (2012) Higher-order functional reactive programming in bounded space. Field, J. and Hicks, M. (eds), *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012* pp. 45–58. ACM.

Mannaa, B. and Møgelberg, R. E. (2018) The clocks they are adjunctions: Denotational semantics for clocked type theory. *3rd International Conference on Formal Structures for Computation and*

Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK pp. 23:1–23:17.
Nakano, H. (2000) A modality for recursion. *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)* pp. 255–266. IEEE Computer Society.
Nilsson, H., Courtney, A. and Peterson, J. (2002) Functional reactive programming, continued. *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02, pp. 51–64. ACM.
Paterson, R. (2001) A new notation for arrows. *ACM SIGPLAN Notices* **36**(10):229–240. 00234.
Ploeg, A. v. d. and Claessen, K. (2015) Practical principled FRP: forget the past, change the future, FRPNow! *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015, pp. 302–314. Association for Computing Machinery. 00019.
Pnueli, A. (1977) The Temporal Logic of Programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* pp. 46–57. IEEE Computer Society.

A Multi-tick calculus

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

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

$$|\Gamma| = \Gamma \quad \text{if } \Gamma \text{ is tick-free}$$
$$\left|\Gamma, \checkmark, \Gamma'\right| = \Gamma^{\square}, \Gamma'$$

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

$$\mathsf{delay}(C[\mathsf{adv}\, t]) \longrightarrow \mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{delay}(C[\mathsf{adv}\, x]) \quad \text{if } t \text{ is not a variable}$$
$$\lambda x.C[\mathsf{adv}\, t] \longrightarrow \mathsf{let}\ y = \mathsf{adv}\, t\ \mathsf{in}\ \lambda x.(C[y])$$

where C is a term with a single occurrence of a hole $[]$ that is not in the scope of delay adv, box, fix, or a lambda abstraction. Formally, C is generated by the following grammar.

$$\begin{aligned} C ::= {} & [] \mid C\,t \mid t\,C \mid \mathsf{let}\ x = C\ \mathsf{in}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ C \mid \langle C, t\rangle \mid \langle t, C\rangle \mid \mathsf{in}_1\, C \mid \mathsf{in}_2\, C \mid \pi_1\, C \mid \pi_2\, C \\ & \mid \mathsf{case}\, C\ \mathsf{of}\ \mathsf{in}_1\, x.s; \mathsf{in}_2\, x.t \mid \mathsf{case}\, s\ \mathsf{of}\ \mathsf{in}_1\, x.C; \mathsf{in}_2\, x.t \mid \mathsf{case}\, s\ \mathsf{of}\ \mathsf{in}_1\, x.t; \mathsf{in}_2\, x.C \\ & \mid C + t \mid t + C \mid \mathsf{into}\, C \mid \mathsf{out}\, C \mid \mathsf{unbox}\, C \end{aligned}$$

We write $C[t]$ to substitute the unique hole $[]$ in C with the term t.

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

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

A.1 Subject reduction

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_{\checkmark} s : \mathsf{Int} \qquad \Gamma \vdash_{\checkmark} t : \mathsf{Int}}{\Gamma \vdash_{\checkmark} s + t : \mathsf{Int}} \qquad \frac{|\Gamma|, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash_{\checkmark} s : A \qquad \Gamma, x : A \vdash_{\checkmark} t : B}{\Gamma \vdash_{\checkmark} \mathsf{let}\, x = s \,\mathsf{in}\, t : B}$$

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

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

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

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

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

Fig. 14. Typing rules of $\lambda_{\checkmark}$.

Proof We proceed by induction on the structure of C.

- $\underline{[]}$: $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} \mathsf{adv}\, t : A$ and Γ' tick-free implies that $\Gamma \vdash_{\checkmark\!\!\checkmark} t : \bigcirc A$. Moreover, given a fresh variable x, we have that $\Gamma, x : \bigcirc A \vdash_{\checkmark\!\!\checkmark} x : \bigcirc A$, and thus $\Gamma, x : \bigcirc A, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} \mathsf{adv}\, x : A$ and Γ'.
- $\underline{C\, s}$: $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[\mathsf{adv}\, t]\, s : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} s : A'$ and $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[\mathsf{adv}\, t] : A' \to A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\checkmark\!\!\checkmark} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[\mathsf{adv}\, x] : A' \to A$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} s : A'$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[\mathsf{adv}\, x]\, s : A$.
- $\underline{s\, C}$: $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[s\, \mathsf{adv}\, t] : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} s : A' \to A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark\!\!\checkmark} C[\mathsf{adv}\, t] : A'$. By induction hypothesis, the latter implies that

there is some B with $\Gamma \vdash_{\mathscr{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} C[\mathsf{adv}\, x] : A'$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} s : A' \to A$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} s\, C[\mathsf{adv}\, x] : A$.

- $\underline{\mathsf{let}\ y = s\ \mathsf{in}\ C}$: $\Gamma, \checkmark, \Gamma' \vdash_{\mathscr{W}} \mathsf{let}\ y = s\ \mathsf{in}\ C[\mathsf{adv}\, t] : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma' \vdash_{\mathscr{W}} s : A'$ and $\Gamma, \checkmark, \Gamma', y : A' \vdash_{\mathscr{W}} C[\mathsf{adv}\, t] : A$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathscr{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma', y : A' \vdash_{\mathscr{W}} C[\mathsf{adv}\, x] : A$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} s : A'$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} \mathsf{let}\ y = s\ \mathsf{in}\ C[\mathsf{adv}\, x] : A$.
- $\underline{\mathsf{let}\ y = C\ \mathsf{in}\ s}$: $\Gamma, \checkmark, \Gamma' \vdash_{\mathscr{W}} \mathsf{let}\ y = C[\mathsf{adv}\, t]\ \mathsf{in}\ s : A$ implies that there is some A' with $\Gamma, \checkmark, \Gamma', y : A' \vdash_{\mathscr{W}} s : A$ and $\Gamma, \checkmark, \Gamma' \vdash_{\mathscr{W}} C[\mathsf{adv}\, t] : A'$. By induction hypothesis, the latter implies that there is some B with $\Gamma \vdash_{\mathscr{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} C[\mathsf{adv}\, x] : A'$. Hence, $\Gamma, x : \bigcirc B, \checkmark, \Gamma', y : A' \vdash_{\mathscr{W}} s : A$, by Lemma A.1, and thus $\Gamma, x : \bigcirc B, \checkmark, \Gamma' \vdash_{\mathscr{W}} \mathsf{let}\ y = C[\mathsf{adv}\, x]\ \mathsf{in}\ s : A$.

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

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

Proof We proceed by induction on the structure of C:

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

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

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

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

- Let $s \longrightarrow t$ be due to congruence closure. Then $\Gamma \vdash_{\mathscr{W}} t : A$ follows by the induction hypothesis. For example, if $s = s_1\, s_2$, $t = t_1\, s_2$ and $s_1 \longrightarrow t_1$, then we know that $\Gamma \vdash_{\mathscr{W}} s_1 : B \to A$ and $\Gamma \vdash_{\mathscr{W}} s_2 : B$ for some type B. By induction hypothesis, we then have that $\Gamma \vdash_{\mathscr{W}} t_1 : B \to A$ and thus $\Gamma \vdash_{\mathscr{W}} t : A$.

- Let $\mathsf{delay}(C[\mathsf{adv}\, t]) \longrightarrow \mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{delay}(C[\mathsf{adv}\, x])$ and $\Gamma \vdash_{\mathcal{W}} \mathsf{delay}(C[\mathsf{adv}\, t]) : A$. That is, $A = \bigcirc A'$ and $\Gamma, \checkmark \vdash_{\mathcal{W}} C[\mathsf{adv}\, t] : A'$. Then by Lemma A.2, we obtain some type B such that $\Gamma \vdash_{\mathcal{W}} t : \bigcirc B$ and $\Gamma, x : \bigcirc B, \checkmark \vdash_{\mathcal{W}} C[\mathsf{adv}\, x] : A'$. Hence, $\Gamma \vdash_{\mathcal{W}} \mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{delay}(C[\mathsf{adv}\, x]) : A$.
- Let $\lambda x.C[\mathsf{adv}\, t] \longrightarrow \mathsf{let}\ y = \mathsf{adv}\, t\ \mathsf{in}\ \lambda x.(C[y])$ and $\Gamma \vdash_{\mathcal{W}} \lambda x.C[\mathsf{adv}\, t] : A$. Hence, $A = A_1 \to A_2$ and $\Gamma, x : A_1 \vdash_{\mathcal{W}} C[\mathsf{adv}\, t] : A_2$. Then, by Lemma A.3, there is some type B such that $\Gamma \vdash_{\mathcal{W}} \mathsf{adv}\, t : B$ and $\Gamma, y : B, x : A_1 \vdash_{\mathcal{W}} C[y] : A_2$. Hence, $\Gamma \vdash_{\mathcal{W}} \mathsf{let}\ y = \mathsf{adv}\, t\ \mathsf{in}\ \lambda x.(C[y]) : A$.

■

A.2 Exhaustiveness

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

Definition A.5.

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

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

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

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

Lemma A.6.

(i) If $\mathsf{delay}\, t \not\longrightarrow$*, then t is weakly* adv*-free.*

(ii) If $\lambda x.\, t \not\longrightarrow$*, then t is strictly* adv*-free.*

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

Lemma A.7. *Let Γ' contain at least one tick, $\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} t : A$, $t \not\longrightarrow$, and t weakly* adv*-free. Then $\Gamma^{\Box}, \Gamma' \vdash_{\mathcal{W}} t : A$*

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

- $\underline{\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \mathsf{adv}\, t : A}$: Then there are Γ_1, Γ_2 such that Γ_2 tick-free, $\Gamma' = \Gamma_1, \checkmark, \Gamma_2$, and $\Gamma, \checkmark, \Gamma_1 \vdash_{\mathcal{W}} t : \bigcirc A$. Since $\mathsf{adv}\, t$ is by assumption weakly adv-free, we know that t is some variable x. Since $\bigcirc A$ is not stable we thus know that $x : \bigcirc A \in \Gamma_1$. Hence, $\Gamma^{\Box}, \Gamma_1 \vdash_{\mathcal{W}} t : \bigcirc A$, and therefore $\Gamma^{\Box}, \Gamma' \vdash_{\mathcal{W}} \mathsf{adv}\, t : A$.
- $\underline{\Gamma, \checkmark, \Gamma' \vdash_{\mathcal{W}} \mathsf{delay}\, t : \bigcirc A}$: Hence, $\Gamma, \checkmark, \Gamma', \checkmark \vdash_{\mathcal{W}} t : A$. Moreover, since $\mathsf{delay}\, t \not\longrightarrow$, we have by Lemma A.6 that t is weakly adv-free. We may thus apply the induction hypothesis to obtain that $\Gamma^{\Box}, \Gamma', \checkmark \vdash_{\mathcal{W}} t : A$. Hence, $\Gamma^{\Box}, \Gamma' \vdash_{\mathcal{W}} \mathsf{delay}\, t : \bigcirc A$.

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

■

Lemma A.8. *Let $\Gamma, \checkmark, \Gamma' \vdash_{\checkmark} t : A$, $t \not\longrightarrow$ and t strictly* adv*-free. Then $\Gamma^{\Box}, \Gamma' \vdash_{\checkmark} t : A$.*
(Note that this Lemma is about $\lambda_{\checkmark}$.)

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

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

■

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

Proposition A.9 (exhaustiveness)**.** *Let* $\Gamma \vdash_{\checkmark}$, $\Gamma \vdash_{\checkmark\!\!\checkmark} t : A$ *and* $t \not\longrightarrow$. *Then* $\Gamma \vdash_{\checkmark} t : A$.

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

- $\dfrac{\Gamma, \checkmark \vdash_{\checkmark\!\!\checkmark} t : A}{\Gamma \vdash_{\checkmark\!\!\checkmark} \mathsf{delay}\, t : \bigcirc A}$:
  We consider two cases:
  - Γ is tick-free: Hence, $\Gamma, \checkmark \vdash_{\checkmark}$ and thus $\Gamma, \checkmark \vdash_{\checkmark} t : A$ by induction hypothesis. Since $|\Gamma| = \Gamma$, we thus have that $\Gamma \vdash_{\checkmark} \mathsf{delay}\, t : \bigcirc A$.
  - $\Gamma = \Gamma_1, \checkmark, \Gamma_2$ and Γ_2 tick-free: By definition, $t \not\longrightarrow$ and, by Lemma A.6, t is weakly adv-free. Hence, by Lemma A.7, we have that $\Gamma_1^{\square}, \Gamma_2, \checkmark \vdash_{\checkmark\!\!\checkmark} t : A$. We can thus apply the induction hypothesis to obtain that $\Gamma_1^{\square}, \Gamma_2, \checkmark \vdash_{\checkmark} t : A$. Since $|\Gamma| = \Gamma_1^{\square}, \Gamma_2$, we thus have that $\Gamma \vdash_{\checkmark} \mathsf{delay}\, t : \bigcirc A$.
- $\dfrac{\Gamma, x : A \vdash_{\checkmark\!\!\checkmark} t : B}{\Gamma \vdash_{\checkmark\!\!\checkmark} \lambda x.t : A \to B}$:
  By induction hypothesis, we have that $\Gamma, x : A \vdash_{\checkmark} t : B$. There are two cases to consider:
  - Γ is tick-free: Then $|\Gamma| = \Gamma$ and we thus obtain that $\Gamma \vdash_{\checkmark} \lambda x.t : A \to B$.
  - $\Gamma = \Gamma_1, \checkmark, \Gamma_2$ with Γ_1, Γ_2 tick-free: Since $t \not\longrightarrow$ by definition and t strictly adv-free by Lemma A.6, we may apply Lemma A.8 to obtain that $\Gamma_1^{\square}, \Gamma_2, x : A \vdash_{\checkmark} t : B$. Since $|\Gamma| = \Gamma_1^{\square}, \Gamma_2$, we thus have that $\Gamma \vdash_{\checkmark} \lambda x.t : A \to B$.
- All remaining cases follow immediately from the induction hypothesis, because all other rules are either the same for both calculi or the $\lambda_{\checkmark}$ typing rule has an additional side condition that Γ have at most one tick, which holds by assumption.

■

A.3 Strong Normalisation

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

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

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

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