Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks

PATRICK BAHR, IT University of Copenhagen, Denmark
CHRISTIAN ULDAL GRAULUND, IT University of Copenhagen, Denmark
RASMUS EJLERS MØGELBERG, IT University of Copenhagen, Denmark

Functional reactive programming (FRP) is a paradigm for programming with signals and events, allowing the user to describe reactive programs on a high level of abstraction. For this to make sense, an FRP language must ensure that all programs are causal, and can be implemented without introducing space leaks and time leaks. To this end, some FRP languages do not give direct access to signals, but just to signal functions.

Recently, modal types have been suggested as an alternative approach to ensuring causality in FRP languages in the synchronous case, giving direct access to the signal and event abstractions. This paper presents *Simply RaTT*, a new modal calculus for reactive programming. Unlike prior calculi, Simply RaTT uses a Fitch-style approach to modal types, which simplifies the type system and makes programs more concise. Echoing a previous result by Krishnaswami for a different language, we devise an operational semantics that safely executes Simply RaTT programs without space leaks.

We also identify a source of time leaks present in other modal FRP languages: The unfolding of fixed points in delayed computations. The Fitch-style presentation allows an easy way to rules out these leaks, which appears not to be possible in the more traditional dual context approach.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Data flow languages**; *Recursion*; • **Theory of computation** → *Operational semantics.*

Additional Key Words and Phrases: Functional reactive programming, Modal types, Synchronous data flow languages, Type systems, Garbage collection

ACM Reference Format:
Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP, Article 109 (August 2019), 27 pages. https://doi.org/10.1145/3341713

1 INTRODUCTION

Reactive programs are programs that engage in an ongoing dialogue with their environment, taking inputs and producing outputs, typically dependent on an internal state. Examples include GUIs, servers, and control software for components in cars, aircraft, and robots. These are traditionally implemented in imperative programming languages using often complex webs of components communicating through callbacks and shared state. As a consequence, reactive programming in imperative languages is error-prone and program behaviour difficult to reason about. This is unfortunate since many of the most safety-critical programs in use today are reactive.

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark, paba@itu.dk; Christian Uldal Graulund, IT University of Copenhagen, Denmark, cgra@itu.dk; Rasmus Ejlers Møgelberg, IT University of Copenhagen, Denmark, mogel@itu.dk.

2475-1421/2019/8-ART109
https://doi.org/10.1145/3341713

The idea of Functional Reactive Programming (FRP) [Elliott and Hudak 1997] is to bring reactive programming into the functional paradigm by providing the programmer with abstractions for describing the dataflow between components in a simple and direct way. At the same time, this should give the usual benefits of functional programming: Modular programming using higher-order functions, and simple equational reasoning. The abstractions provided by the early FRP languages were *signals* and *events*: A signal of type A is a time-varying value of type A, and an event of type A is a value of type A appearing at some point in time. The notion of time is abstract, but can, depending on the application, be thought of as either discrete or continuous.

For such high-level abstractions to make sense, the language designer must ensure that all programs can be executed in an efficient way. A first problem is ensuring *causality*, i.e., the property that the value of output signals at a given time only depends on the values read from input signals before or at that time. For example, implementing signals in the discrete time case simply as streams will break this abstraction, as there are many non-causal functions from streams to streams. Another issue is *time leaks*, i.e., the problem of programs exhibiting gradually slower response time, typically due to intermediate values being recomputed whenever output is needed. The related notion of *space leaks* is the problem of programs holding on to memory while continually allocating more until they eventually run out of memory.

A good language for FRP should only allow programmers to write causal functions. On the other hand, in expressive programming languages some of the responsibility for avoiding the problems of space and time leaks must be left to the programmer. For example, if the language has linked lists, a programmer could write a function that stores all input in a list, leading to a space leak. We will refer to this as an *explicit* space leak, since it can be detected from the code. A good FRP language should avoid *implicit* space and time leaks, i.e., leaks that are caused by the language implementation, and so are out of the programmers control.

Due to these concerns, newer libraries and languages for FRP do not give the programmer direct access to signal and event types. For example, Arrowised FRP [Nilsson et al. 2002] has a primitive notion of signal functions and provides combinators for combining these to construct dataflow networks statically, along with switching operators for dynamically changing these networks. This approach sacrifices some of the simplicity and flexibility of the original suggestions for FRP, and the switching combinators have an ad hoc flavour. Moreover, to the best of our knowledge, no strong guarantees concerning space or time leaks have been proved in this setting.

1.1 Modal FRP Calculi

Recently, a number of authors ([Jeffrey 2012, 2014; Jeltsch 2013; Krishnaswami 2013; Krishnaswami and Benton 2011]) have suggested using modal types for functional reactive programming. These all work in the synchronous case of time being given by a global clock. With this assumption, the resulting languages can be thought of as extensions of synchronous dataflow languages such as Lustre [Caspi et al. 1987], and Lucid Synchrone [Pouzet 2006] with higher-order functions and operations for dynamically changing the dataflow network. This restricted setting covers many applications of FRP, and in this paper we shall restrict ourselves to that as well. Since continuous time can be simulated by discrete time (see Section 7), we will further restrict ourselves to discrete time.

Under the assumption of a global discrete clock a signal is simply a stream. Causality is ensured by using a modal type operator $\bigcirc$ to encode the notion of a time step in types: A value of type $\bigcirc A$ is a computation returning a value of type A in the next time step. Using $\bigcirc$, one can describe the streams corresponding to signals as a type satisfying the type isomorphism $\mathrm{Str}(A) \cong A \times \bigcirc\mathrm{Str}(A)$, capturing the fact that the tail of the stream is only available in the next time step. Streams and

programs processing streams can be defined recursively using the guarded fixed point combinator of Nakano [2000] taking input of type $\bigcirc A \to A$ and producing elements of type A as output.

The most advanced programming language of this kind, in terms of operational semantics with run-time guarantees, is that of Krishnaswami [2013]. This language extends the simply typed lambda calculus with two modal type operators: The $\bigcirc$ mentioned above, as well as one for classifying stable, i.e., time-invariant data. Unlike the arrowised approach to FRP, Krishnaswami's calculus gives direct access to streams as a data type which can even be nested to give streams of streams. Other important data types, such as events can be encoded using *guarded recursive types*, a concept also stemming from Nakano [2000].

Krishnaswami's calculus has an operational semantics for evaluating terms in each step of the global clock, and this can be extended to a step-by-step evaluation of streams. The language is total in the sense that each step evaluates to a value in finite time (a property often referred to as *productivity*). The operational semantics evaluates by storing delayed computations on a heap, and Krishnaswami shows that all heap data can be safely garbage collected after each evaluation step, effectively guaranteeing the absence of (implicit) space leaks.

1.2 Fitch-style Modal Calculi

Like most modal calculi, Krishnaswami's calculus uses let-expressions to program with modalities. This affects the programming style: Many programs consist of a long series of unpacking statements, essentially giving access to the values produced by delayed computations in the next time step, followed by relatively short expressions manipulating these. While this can be to a large extent be dealt with using syntactic sugar, it has a more fundamental problem, which is harder to deal with: It complicates equational reasoning about programs. This is an important issue, since simple equational reasoning is supposed to be one of the benefits of functional programming. Our long-term goal is to design a dependent type theory for reactive programming in which programs have operational guarantees like the ones proved by Krishnaswami, and where program specifications can be expressed using dependent types. Introducing let-expressions in terms will lead to let-expressions also in the types, which is a severe complication of the type theory.

Fitch-style modal calculi [Clouston 2018; Fitch 1952] are an alternative approach to modal types not using let-expressions. Instead, elements of modal types are constructed by abstracting tokens from the context, and modal operators are likewise eliminated by placing tokens in the context. Recent research in guarded dependent type theory [Bahr et al. 2017; Clouston et al. 2018] has shown the benefit of this approach also for dependent types. Guarded dependent type theory is an extension of Martin-Löf type theory [Martin-Löf and Sambin 1984] with a delay modality reminiscent of the $\bigcirc$ used in modal FRP together with Nakano's fixed point combinator also mentioned above. In this setting, the token used in the Fitch-style approach is thought of as a 'tick' – evidence that time has passed – which can be used to open a delayed computation. Using ticks, one can prove properties of guarded recursive programs in a compellingly simple way.

1.3 Simply RaTT

In this paper, we present Simply Typed Reactive Type Theory (Simply RaTT) a simply typed calculus for reactive programming based on the Fitch-style approach to modal types. This is a first step towards our goal of a dependently typed theory for reactive programming (RaTT), but already the simply typed version offers several benefits over existing approaches. Compared to Krishnaswami's calculus, Simply RaTT has a significantly simpler type system. The Fitch-style approach eliminates the need for the qualifiers 'now', 'later' and 'stable' used in Krishnaswami's calculus on variables and term judgements. Similar (but not quite the same) qualifiers can be derived from the position of variables relative to tokens in contexts in Simply RaTT. Moreover, we eliminate the need for

allocation tokens, a technical tool used by Krishnaswami to control heap allocation. This, together with the Fitch-style typing rules makes programs shorter and (we believe) more readable than in Krishnaswami's calculus.

Compared to the standard approach to modal types, the Fitch-style used here is based on a shift in time-dependence. Whereas terms in Krishnaswami's language can look into the future (since now-terms can depend on later-variables), terms in Simply RaTT can only look into the past (since later-expressions can depend on now-variables). This explains how let-expressions are eliminated: There is no need to refer to the values produced in the future by delayed computations. Instead, Simply RaTT allows delayed computations from the past to be run in the present.

We prove a garbage collection result similar to that proved by Krishnaswami, and show how this can be used to construct a safe evaluation strategy for stream transducers written in our language. Input to stream transducers are treated as delayed computations, and therefore stored in a heap and garbage collected in the next time step.

We also identify and eliminate a source of time leaks present in previous approaches. This is best illustrated by the following two implementations of the stream of natural numbers written in Haskell-notation:

$$leakyNats = 0 :: map\ (+1)\ leakyNats \qquad nats = from\ 0 \ \textbf{where}\ from\ n = n :: from\ (n+1)$$

On most machines (some compilers may use clever techniques to detect this problem), the evaluation of the nth element of *leakyNats* will not use the previously computed values, but instead compute it using n successive applications of *suc*, resulting in a time leak. This is indeed what happens on Krishnaswami's machine and also the machine of this paper. Contrary to that, the *nats* example uses an internal state declared explicitly in the type of *from* to maintain a constant evaluation time for each step. In this paper we identify the source of the time leak to be the ability to unfold fixed points in delayed computations, and use this to eliminate examples such as *leakyNats* in typing. The ability to control when unfolding of fixed points are allowed relies crucially on the Fitch-style presentation, and it is very unclear whether a similar restriction can be added to the traditional dual context presentation.

The calculus is illustrated through examples showing how to implement a small FRP library as well as how to simulate the most basic constructions of Lustre in Simply RaTT. Examples are also used to illustrate our abstract machine for evaluating streams and stream transducers.

1.4 Overview of Paper

The paper is organised as follows: Section 2 gives an overview of the language introducing the main concepts and their intuitions through examples. Section 3 defines the operational semantics, including the evaluation of stream transducers and states the garbage collection results for these. Section 4 shows how to implement a small library for reactive programming in Simply RaTT and Section 5 shows how to encode the most basic constructions of the synchronous dataflow language Lustre in Simply RaTT. Section 6 sketches the proof of our garbage collection result. The metatheory presented in Section 6 has been fully formalised in the accompanying Coq proofs. Finally, Section 7 describes related work and Section 8 concludes and describes future work.

2 SIMPLY RATT

This section gives an overview of the Simply RaTT language. The complete formal description of the syntax of the language, and in particular the typing rules, can be found in Figure 2, Figure 3, and Figure 4.

$\Gamma \vdash t : A$ (a) Initial judgement

$\Gamma, \sharp, \Gamma_N \vdash t : A$ (b) Now judgement

$\Gamma, \sharp, \Gamma_N, \checkmark, \Gamma_L \vdash t : A$ (c) Later judgement

Fig. 1. The different type judgement forms. In these, the contexts Γ, Γ_N and Γ_L are assumed to be token-free and contain variables referred to as initial variables, now-variables and, later-variables.

The type system of Simply RaTT extends that of the simply typed lambda calculus with two modal type operators: $\bigcirc$ for classifying delayed computations, and $\Box$ for classifying stable computations, i.e., computations that can be performed safely at any time in the future. We start by describing the constructions for $\bigcirc$.

Data of type $\bigcirc A$ are *computations* that produce data of type A in the next time step. To perform such a computation we must wait a time step, as represented in typing judgements by the addition of a $\checkmark$ (pronounced 'tick') in the context. More precisely, the typing rule for eliminating $\bigcirc$ states that if $\Gamma \vdash t : \bigcirc A$ then $\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}(t) : A$. The $\checkmark$ in the context of $\mathsf{adv}(t)$ should be thought of as separating variables in time: Those in Γ are available one time step before those in Γ'. Since there can be at most one $\checkmark$ in a context, we will refer to these times as 'now' and 'later'. The typing assumption on t states that it has type $\bigcirc A$ now, and the conclusion states that $\mathsf{adv}(t)$ has type A later. The constructor for $\bigcirc A$ states that if $\Gamma, \checkmark \vdash t : A$, i.e., if t has type A later, but depends only on variables available now, then it can be turned into a *thunk* $\mathsf{delay}(t)$ of type $\bigcirc A$ now.

Note that terms in 'later' judgements can refer to variables available now as well as later, but 'now' judgements can only refer to variables available now. This separates the Fitch-style approach of Simply RaTT from the traditional dual context approach to calculi with modalities, such as Krishnaswami's [2013] modal calculus for reactive programming. The latter also has a distinction between 'later' and 'now', but the time dependencies work the opposite way: A later-judgement can only depend on later-variables, whereas a now-judgement can depend on both now- and later-variables.

Data of type $\Box A$ are time invariant *computations* that produce data of type A. That is, these computations can be executed safely at any time in the future. To allow time invariant computations to depend on initial data, that is, data available before the reactive program starts executing, contexts may contain a $\sharp$ separating the context into *initial variables* (those to the left of $\sharp$) and *temporal variables* to the right of $\sharp$. There can be at most one $\sharp$ in a context, and $\Gamma, \checkmark$ is only well-formed if there is a $\sharp$ in Γ. Thus $\checkmark$ separates the temporal variables into now and later. We refer collectively to $\checkmark$ and $\sharp$ as *tokens*. Judgements in a token-free context is referred to as an initial judgement. The three kinds of judgements are summarised in Figure 1.

If $\Gamma, \sharp \vdash t : A$ then t does not depend on any temporal data, and can thus be thunked to a time invariant computation $\Gamma \vdash \mathsf{box}(t) : \Box A$ to be run at a later time. The typing rule for eliminating $\Box$ states that if $\Gamma \vdash t : \Box A$ and Γ' is token-free, then $\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}(t) : A$. The restriction on Γ' means that we can only run the time invariant computation t now, not later. This may seem to contradict the intuition for $\Box A$ given above, but is needed to rule out certain time leaks as we shall see below. Time invariant computations can still be run at arbitrary times in the future through the use of fixed points.

Both these modal type operators have restricted forms of applicative actions. In the case of $\bigcirc$, if $\Gamma \vdash t : \bigcirc(A \to B)$ and $\Gamma \vdash u : \bigcirc A$ then $\Gamma \vdash t \circledast u : \bigcirc B$ is defined as

$$t \circledast u = \mathsf{delay}(\mathsf{adv}(t)(\mathsf{adv}(u))).$$

Note that this is only well-typed if Γ contains $\sharp$ but not $\checkmark$, since the subterm $\mathsf{adv}(f)(\mathsf{adv}(x))$ must be typed in context $\Gamma, \checkmark$, and by the restrictions mentioned above, this is only a well-formed context if $\sharp$ is the only token in Γ. Similarly, if $\Gamma \vdash t : \Box(A \to B)$ and $\Gamma \vdash u : \Box A$ then $\Gamma \vdash t \circledast u : \Box B$ is defined as $\mathsf{box}(\mathsf{unbox}(t)(\mathsf{unbox}(u)))$. As above, this is only well-typed if Γ is token-free. Note that neither $\Box$ nor $\bigcirc$ are applicative functors in the sense of McBride and Paterson [2008], since there are generally no maps $A \to \Box A$, nor $A \to \bigcirc A$. The former would force computations to be stable, and the latter would push data into the future, which is generally unsafe as it can lead to space leaks. This restriction is enforced in the type theory in the variable introduction rule, which does not allow variables to be introduced over tokens. As a consequence, weakening of typing judgements with tokens is not admissible. An exception to this exists for the *stable* types, as we shall see below.

2.1 Fixed Points

Reactive programs can be defined recursively using a fixed point combinator. To ensure productivity and causality, the recursion variable must be a delayed computation. Precisely, the rule for fixed points state that if $\Gamma, \sharp, x : \bigcirc A \vdash t : A$ then $\Gamma \vdash \mathsf{fix}\, x.t : \Box A$. These guarded recursive fixed points can be used to program with guarded recursive types such as guarded recursive streams $\mathsf{Str}(A)$ satisfying the type isomorphism $\mathsf{Str}(A) \cong A \times \bigcirc\mathsf{Str}(A)$. Terms of this type compute to an element in A (the head) now, and a delayed computation of a tail. We will use :: as infix notation for the right to left direction of the isomorphism, i.e., $t :: u$ is a shorthand for $\mathsf{into}\,\langle t, u\rangle$. Given $t : A$ and $u : \bigcirc\mathsf{Str}(A)$, we thus have $t :: u : \mathsf{Str}(A)$.

As a simple example of a recursive definition, the stream of all zeros can be defined as

$$\mathit{zeros} = \mathsf{fix}\ x\,.\,0 :: x : \Box\ (\mathsf{Str}\ (\mathsf{Nat}))$$

Note that fixed points are time invariant in the sense of having a type of the form $\Box A$. This is because they essentially need to call themselves in the future. For this reason, their definition cannot depend on temporal data, as can be seen from the typing rule, since x must be the only temporal variable in t.

As a second example of a recursively defined function, we define a map function for guarded streams. This should take a function $A \to B$ as input and a stream of type $\mathsf{Str}(A)$ and produce a stream of type $\mathsf{Str}(B)$. Since the input function will be called repeatedly at all future time steps it needs to be time-invariant, and can be defined as:

$$\begin{aligned}&\mathit{map} : \Box\ (A \to B) \to \Box\ (\mathsf{Str}\ A \to \mathsf{Str}\ B)\\ &\mathit{map} = \lambda f\,.\,\mathsf{fix}\ x\,.\,\lambda \mathit{as}\,.\,\mathsf{unbox}\ f\ (\mathsf{head}\ \mathit{as}) :: x \circledast \mathsf{tail}\ \mathit{as}\end{aligned}$$

where head and tail compute the head and the tail of a stream, respectively.

For readability we introduce the following syntax for defining fixed points such as *map*:

$$\mathit{map}\ f\ \sharp\ (a :: \mathit{as}) = \mathsf{unbox}\ f\ \ a :: \mathit{map}\ f \circledast \mathit{as}$$

This should be read as defining the term to the left of $\sharp$ as a fixed point and in particular it allows us to write pattern matching in a simple way. When type checking the right-hand side of this definition, $\mathit{map}\ f$ should be given type $\bigcirc(\mathsf{Str}(A) \to \mathsf{Str}(B))$ because it represents the recursion variable. Any such definition can be translated syntactically to our core language in a straightforward manner: Pattern matching is translated to the corresponding elimination forms (π_i, case, out) and the recursion syntax with $\sharp$ is translated to fix.

The type of guarded streams defined above is just one example of a guarded recursive type. Simply RaTT includes a construction for general recursive types $\mu\alpha.A$ satisfying type isomorphisms of the form $\mu\alpha.A \cong A[\bigcirc(\mu\alpha.A)/\alpha]$. In these α can appear everywhere in A, including non-strictly positive and even negative positions. Another example of a guarded recursive type is that of events

defined as $\mathsf{Ev}(A) = \mu\alpha.A + \alpha$, and thus satisfying $\mathsf{Ev}(A) \cong A + \bigcirc\mathsf{Ev}(A)$. Streams and events form the building blocks of functional reactive programming. Similarly to streams, one can define a map function for events using fixed points as follows

$$
\begin{aligned}
&\mathit{map} : \Box\,(A \to B) \to \Box\,(\mathsf{Ev}\ A \to \mathsf{Ev}\ B)\\
&\mathit{map}\ f\ \sharp\,(\mathsf{wait}\ \mathit{eva}) = \mathsf{wait}\ (\mathit{map}\ f \circledast \mathit{eva})\\
&\mathit{map}\ f\ \sharp\,(\mathsf{val}\ a) \quad\ = \mathsf{val}\ (f\ a)
\end{aligned}
$$

where we write $\mathsf{val}\,t$ and $\mathsf{wait}\,t$ instead of $\mathsf{into}\,(\mathsf{in}_1\,t)$ and $\mathsf{into}\,(\mathsf{in}_2\,t)$, respectively.

2.2 Stable Types

Next we show how to define the stream of natural numbers using a helper function mapping a natural number n to the stream $(n, n+1, n+2, \dots)$. A first attempt at defining *from* could look as follows:

$$
\begin{aligned}
&\mathit{from} : \Box\,(\mathsf{Nat} \to \mathsf{Str}\,(\mathsf{Nat}))\\
&\mathit{from}\,\sharp\,n = n :: \mathit{from} \circledast \mathsf{delay}\ (n+1)
\end{aligned}
$$

is not well typed, because to type $\mathsf{delay}(n+1)$ the term $n+1$ must have type Nat *later*, but n is a *now*-variable. The number n therefore must be kept for the next time step, an operation that generally is unsafe, because general values can have references to temporal data. For example, a value of type $\bigcirc\mathsf{Str}(A)$ in our machine is a reference to the tail of a stream, which could be an input stream. Allowing such values to be kept for the next step can lead the machine to store input data indefinitely, causing space leaks. Similarly, values of function types can contain references to time dependent data in closures and should therefore not be kept. On the other hand, a value of type natural numbers cannot contain such references and so can safely be kept for the next time step. We say that Nat is a *stable* type, and a grammar for these stable types is given in Figure 3. Data of stable type can be kept one time step using the construction progress which allows a now-judgement of the form $\Gamma \vdash t : A$ to be transformed to a later judgement of the form $\Gamma, \checkmark, \Gamma' \vdash \mathsf{progress}\,t : A$ if Γ contains a $\sharp$ and no $\checkmark$ and if A is stable. In our operational semantics, $\mathsf{progress}\,t$ evaluates by evaluating t to a value now pushing the result to the future. Postponing the evaluation of t would be unsafe, since terms of stable types, unlike values of stable types, can refer to temporal data. Similarly, promote can be used to make stable initial data available in temporal judgements.

We introduce the constructions $\odot$, defined as $t \odot u = \mathsf{delay}(\mathsf{adv}(t)(\mathsf{progress}\,u))$, and $\boxdot$, defined as $t \boxdot u = \mathsf{box}(\mathsf{unbox}(t)(\mathsf{promote}\,u))$, with derived typing rules

$$
\frac{\Gamma \vdash t : \bigcirc(A \to B) \qquad \Gamma \vdash u : A \qquad \Gamma, \checkmark \vdash \qquad A\ \mathsf{stable}}{\Gamma \vdash t \odot u : \bigcirc B}
$$

$$
\frac{\Gamma \vdash t : \Box(A \to B) \qquad \Gamma \vdash u : A \qquad \Gamma, \sharp \vdash \qquad A\ \mathsf{stable}}{\Gamma \vdash t \boxdot u : \Box B}
$$

Using this, *from* and *nats* can be defined as follows

$$
\begin{aligned}
&\mathit{from} : \Box\,(\mathsf{Nat} \to \mathsf{Str}\ \mathsf{Nat}) && \mathit{nats} = \Box\,(\mathsf{Str}\ \mathsf{Nat})\\
&\mathit{from}\,\sharp\,n = n :: \mathit{from} \odot (n+1) && \mathit{nats} = \mathit{from} \boxdot 0
\end{aligned}
$$

Many programming languages would also allow *nats* to be defined directly as fixed point as $\mathit{leakyNats} = 0 :: \mathit{map}\ (+1)\ \mathit{leakyNats}$. In Simply RaTT, however, such a definition would not be well typed, because the term $\mathit{map}(\mathsf{box}(+1))$ of type $\Box(\mathsf{Str}(\mathsf{Nat}) \to \mathsf{Str}(\mathsf{Nat}))$ would have to be unboxed

in a context with a ✓ in order to type a term like

$$leakyNats\,\sharp\,=\,0\,::\,\mathsf{delay}(\mathsf{unbox}(map(\mathsf{box}(+1))))\circledast leakyNats$$

and this is not allowed according to the typing rule for unbox. We believe such a definition should be ruled out because it leads to time leaks as explained in the introduction. This indeed happens on the machined described in Section 3 as well as the machine of Krishnaswami [2013].

The time leak in the *leakyNats* example above happens because the fixed point definition of *map* is unfolded in a delayed term, allowing the term to be evaluated to grow for each iteration. In the *nats* example, on the other hand, the recursive definition uses a state, namely the input to *from*, to avoid repeating computations. Moreover, this state usage is essentially declared in the type of *from*.

For similar reasons, the *scary_const* example of Krishnaswami [2013] in which all data from an input stream is kept indefinitely by explicitly storing it in a stream of streams cannot be typed in Simply RaTT. An implementation of *scary_const* in Simply RaTT would require an explicit state that stores all previous elements from the input stream. That could be achieved by extending the language to include a list type List A, and defining that List A is stable if A is. The fact that the memory usage of *scary_const* is unbounded is then reflected by the fact that the state of type List A that is needed for *scary_const* is unbounded in size.

Note that we make crucial use of the Fitch-style presentation to rule out *leakyNats*. In the more traditional dual context approach of Krishnaswami [2013], it does not seem possible to have a similar restriction on unfolding of fixed points. The difference is that Simply RaTT "remembers" when we are under a delay whereas that information is lost in the system by Krishnaswami [2013]. In the example of *leakyNats*, the leak stems from the call of *map* in the tail, which in Krishnaswami's system is typed as a regular now judgement, and thus cannot be prevented.

2.3 Function Types

The operational semantics of Simply RaTT uses a heap for delayed computations as well as input streams. The operation delay(t) stores the computation t on the heap and adv retrieves a delayed computation from the heap and evaluates it. In this sense, delay and adv can be understood as computational effects.

Our main result (Theorem 6.3) states that delayed computations and input data on the heap can be safely garbage collected after each computation step. This result relies crucially on the property that open terms typed in now-judgements cannot retrieve delayed computations from the heap. One reason for this is that such terms can not contain adv unless under delay. To maintain this invariant also for function calls, function types $A \to B$ are restricted to functions with no retrieve effects. For this reason, functions may not be constructed in a later-judgement. Later-variables can still be used for case-expressions, and so are included in Simply RaTT. The language could be extended with an extra function type with read effects, constructed by abstracting later-variables in later-judgements, but we found no use for this in our examples.

For example, we can define a function reading an input stream and returning a stream of functions as follows

$$\begin{aligned} &f : \Box\,(\mathsf{Str}\;\mathsf{Nat} \to \mathsf{Str}\;(\mathsf{Nat} \to \mathsf{Nat})) \\ &f = map\;(\mathsf{box}\;(\lambda n\,.\,\lambda x\,.\,n + x)) \end{aligned}$$

and apply streams of functions as follows

$$\begin{aligned} &strApp : \Box\,(\mathsf{Str}\;(A \to B) \to \mathsf{Str}\;A \to \mathsf{Str}\;B) \\ &strApp\,\sharp\,(f :: fs)\;(a :: as) = f\;a :: strApp \circledast fs \circledast as \end{aligned}$$

$$
\begin{array}{lll}
\text{Types} & A, B & ::= A \mid 1 \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \bigcirc A \mid \Box A \mid \mu\alpha.A \\
\text{Values} & v, w & ::= \langle\rangle \mid \bar{n} \mid \lambda x.t \mid \langle v, w\rangle \mid \mathsf{in}_i\, v \mid \mathsf{box}\, t \mid \mathsf{into}\, v \mid \mathsf{fix}\, x.t \mid l \\
\text{Terms} & s, t & ::= \langle\rangle \mid \bar{n} \mid \lambda x.t \mid \langle s, t\rangle \mid \mathsf{in}_i\, t \mid \mathsf{box}\, t \mid \mathsf{into}\, t \mid \mathsf{fix}\, x.t \mid l \mid x \mid t_1\, t_2 \mid t_1 + t_2 \mid \mathsf{adv}\, t \\
& & \quad \mid \mathsf{delay}\, t \mid \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 \mid \mathsf{unbox}\, t \mid \mathsf{progress}\, t \mid \mathsf{promote}\, t \mid \mathsf{out}\, t
\end{array}
$$

Fig. 2. Syntax.

Well-formed types $\Theta \vdash A : \mathsf{type}$

$$
\frac{\alpha \in \Theta}{\Theta \vdash \alpha : \mathsf{type}} \qquad \frac{}{\Theta \vdash 1 : \mathsf{type}} \qquad \frac{}{\Theta \vdash \mathsf{Nat} : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \times B : \mathsf{type}}
$$

$$
\frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A + B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \to B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \bigcirc A : \mathsf{type}}
$$

$$
\frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \Box A : \mathsf{type}} \qquad \frac{\Theta, \alpha \vdash A : \mathsf{type}}{\Theta \vdash \mu\alpha.A : \mathsf{type}}
$$

Well-formed contexts $\Gamma \vdash$

$$
\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash \qquad \vdash A : \mathsf{type}}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \qquad \mathsf{token\text{-}free}(\Gamma)}{\Gamma, \sharp \vdash} \qquad \frac{\Gamma \vdash \qquad \mathsf{tick\text{-}free}(\Gamma) \qquad \sharp \notin \Gamma}{\Gamma, \checkmark \vdash}
$$

Stable types $A\ \mathsf{stable}$

$$
\frac{}{1\ \mathsf{stable}} \qquad \frac{}{\mathsf{Nat}\ \mathsf{stable}} \qquad \frac{}{\Box A\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A \times B\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A + B\ \mathsf{stable}}
$$

Fig. 3. Context and type formation rules for Simply RaTT.

On the other hand, allowing lambda abstraction of later-variables would type the following (rather contrived) stream definition *leaky*, which breaks the safety of the garbage collection strategy:

$$
\begin{array}{l}
\mathit{leaky}' : \Box\,((1 \to \mathsf{Bool}) \to \mathsf{Str}\ \mathsf{Bool}) \\
\mathit{leaky}'\ \sharp\ p = \mathsf{true} :: \mathsf{delay}\ (\mathsf{adv}\ (\mathbf{if}\ (p\ \langle\rangle)\ \mathbf{then}\ \mathit{leaky}'\ \mathbf{else}\ \mathit{leaky}') \\
\qquad\qquad\qquad\qquad (\lambda x\,.\,\mathsf{head}\ (\mathsf{adv}\ \mathit{leaky}'\ (\lambda y\,.\,\mathsf{true}))))
\end{array}
$$

$$
\begin{array}{l}
\mathit{leaky} : \Box\ (\mathsf{Str}\ \mathsf{Bool}) \\
\mathit{leaky} = \mathsf{box}\ (\mathsf{unbox}\ \mathit{leaky}\ (\lambda x\,.\,\mathsf{true}))
\end{array}
$$

In particular, in the definition of *leaky*′ we use adv on the recursive call of *leaky*′ inside a function. The problem here is that the function body only gets evaluated when applied to an argument. However, this application happens too late – at a time where the recursive call to *leaky*′ has already been garbage collected (cf. Section 3.4).

3 OPERATIONAL SEMANTICS

Following the idea of Krishnaswami [2013], we devise an operational semantics for Simply RaTT that is free of space leaks *by construction*. To this end, the operational semantics is defined in terms

$$\frac{\Gamma, x : A, \Gamma' \vdash \qquad \text{token-free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash s : \mathsf{Nat} \qquad \Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash s + t : \mathsf{Nat}}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \text{tick-free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B}$$

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

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

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

$$\frac{\Gamma \vdash t : A \qquad \Gamma, \checkmark, \Gamma' \vdash \qquad A\ \mathsf{stable}}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{progress}\, t : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, \sharp, \Gamma' \vdash \qquad A\ \mathsf{stable}}{\Gamma, \sharp, \Gamma' \vdash \mathsf{promote}\, t : A}$$

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

Fig. 4. Typing rules of Simply RaTT.

of a machine that has access to a store consisting of up to two separate heaps: A 'now' heap η_N from which we can retrieve delayed computations, and a 'later' heap η_L where we can store computations that should be performed in the next time step. Once the machine advances to the next time step, it will delete the 'now' heap η_N and the 'later' heap η_L will become the new 'now' heap. Thus the problem of proving the absence of space leaks is reduced to the problem of soundness, i.e., that well-typed programs never get stuck.

3.1 Term Semantics

The operational semantics of terms is presented in Figure 5. Given a term t together with a store σ, we write $\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$ to denote that the machine evaluates t in the context of σ to a value v and produces an updated store σ'. Importantly, a store σ can take on three different forms: It may contain no heap, written $\sigma = \bot$; it may consist of one heap η_L, written $\sigma = \sharp\eta_L$; or it may consist of two heaps η_N and η_L, written $\sigma = \sharp\eta_N\checkmark\eta_L$. These different forms of stores enforce effective restrictions on when the machine is allowed to store or retrieve delayed computations. If $\sigma = \bot$, then computations may neither be stored nor retrieved. If $\sigma = \sharp\eta_L$, then computations may be stored in η_L to be retrieved in the next time step. And if $\sigma = \sharp\eta_N\checkmark\eta_L$, computations may be stored in η_L as well as retrieved from η_N. Heaps themselves are simply finite mappings from *heap locations* to terms.

Given a store σ that is not $\bot$, i.e., it is either of the form $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$, the machine can store delayed computations on the 'later' heap η_L. To this end, we use the notation $\mathsf{later}(\sigma)$ to refer to η_L, and given $l \notin \mathrm{dom}(\eta_L)$, we write $\sigma, l \mapsto t$ for the store $\sharp(\eta_L, l \mapsto t)$ or $\sharp\eta_N\checkmark(\eta_L, l \mapsto t)$,

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle u;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle u';\sigma''\rangle}{\langle\langle t,t'\rangle;\sigma\rangle \Downarrow \langle\langle u,u'\rangle;\sigma''\rangle}$$

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

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

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

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \overline{m};\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle \overline{n};\sigma''\rangle}{\langle t+t';\sigma\rangle \Downarrow \langle \overline{m+n};\sigma''\rangle} \qquad \frac{\sigma\neq\bot \qquad l=\mathsf{alloc}\,(\sigma)}{\langle \mathsf{delay}\ t;\sigma\rangle \Downarrow \langle l;\sigma, l\mapsto t\rangle}$$

$$\frac{\langle t;\sharp\eta_N\rangle \Downarrow \langle l;\sharp\eta'_N\rangle \qquad \langle \eta'_N(l);\sharp\eta'_N\checkmark\eta_L\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{adv}\ t;\sharp\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\sigma'\rangle} \qquad \frac{\langle t;\bot\rangle \Downarrow \langle v;\bot\rangle \qquad \sigma\neq\bot}{\langle \mathsf{promote}\ t;\sigma\rangle \Downarrow \langle v;\sigma\rangle}$$

$$\frac{\langle t;\sharp\eta_N\rangle \Downarrow \langle v;\sharp\eta'_N\rangle}{\langle \mathsf{progress}\ t;\sharp\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\sharp\eta'_N\checkmark\eta_L\rangle} \qquad \frac{\langle t;\bot\rangle \Downarrow \langle \mathsf{box}\ t';\bot\rangle \qquad \langle t';\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \sigma\neq\bot}{\langle \mathsf{unbox}\ t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

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

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

Fig. 5. Big-step operational semantics.

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

Fig. 6. Small-step operational semantics for stream unfolding and stream processing.

respectively. In turn, $\eta_L, l\mapsto t$ denotes the heap obtained by extending η_L with a new mapping $l\mapsto t$. To allocate a fresh heap locations, we assume a function $\mathsf{alloc}\,(\cdot)$ that takes a store $\sigma\neq\bot$ and returns a heap location l such that $l\notin \mathsf{dom}\,(\mathsf{later}(\sigma))$. That is, given $l=\mathsf{alloc}\,(\sigma)$, we can form the new store $\sigma, l\mapsto t$ without overwriting any mappings that are present in σ.

As the notation suggests, there is a close correspondence between the shape of a context Γ and the shape of a store σ. Terms typable in an initial judgement (cf. Figure 1) can be executed safely with a store $\bot$ – they need not store nor retrieve delayed computations. Terms typable in a now judgement can be executed safely with a store $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$ – they may store delayed

computations in η_L, but need not retrieve delayed computations. And finally, terms typable in a later judgement can be executed safely in a store of the form $\sharp\eta_N\checkmark\eta_L$ – they may retrieve delayed computations from η_N.

This intuition of the capabilities of the different stores can be observed directly in the semantics for delay and adv: For $\mathsf{delay}\, t$ to evaluate, the machine expects a store that is not $\bot$, i.e., a store $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$. Then the machine allocates a fresh heap location l in the heap η_L and stores t in it. This corresponds to the fact that $\mathsf{delay}\, t$ can only be typed in a now judgement. Conversely, $\mathsf{adv}\, t$ requires the store to be of the form $\sharp\eta_N\checkmark\eta_L$ so that t can be evaluated safely with the store $\sharp\eta_N$ to a heap location l, which either already existed in η_N or was allocated when evaluating t. In either case, the delayed computation stored at heap location l is retrieved and executed. The combinator progress with its typing rule similar to that of adv, also has similar operational behaviour in terms of how it interacts with the store.

Fixed points are evaluated when a term t that evaluates to a value of the form $\mathsf{fix}\, x.t'$ is unboxed. For a general fixed point combinator, we would expect that $\mathsf{fix}\, x.t'$ unfolds to $t'[\mathsf{fix}\, x.t'/x]$. In our setting, the types dictate that $\mathsf{fix}\, x.t'$ should rather unfold to $t'[\mathsf{delay}(\mathsf{unbox}(\mathsf{fix}\, x.t'))/x]$, because x has type $\bigcirc A$ and $\mathsf{fix}\, x.t'$ has type $\Box A$. This is close to the behaviour of our machine (and would in fact be a safe alternative definition). Instead, however, the machine anticipates that the term allocates a mapping $l \mapsto \mathsf{unbox}(\mathsf{fix}\, x.t')$ on the store and evaluates to that heap location l. Therefore, the machine evaluates the fixed point by allocating a mapping $l \mapsto \mathsf{unbox}(\mathsf{fix}\, x.t')$ on the store right away and evaluating $t'[l/x]$ subsequently.

3.2 Stream Semantics

The careful distinction between a 'now' heap η_N and a 'later' heap η_L is crucial in order to avoid *implicit* space leaks. After the machine has evaluated a term t to a value v and produced a store of the form $\sharp\eta_N\checkmark\eta_L$, we can safely garbage collect the entire heap η_N and compute the next step with the store $\sharp\eta_L\checkmark$. For example, if the original term t was of type $\mathsf{Str}(\mathsf{Nat})$, then its value v will be of the form $\overline{n} :: l$, where n is the head of the stream and l is a heap location that points to the delayed computation that computes the tail of the steam. The tail of the stream can then be safely computed by evaluating $\mathsf{adv}\, l$ with the store $\sharp\eta_L\checkmark$, i.e. with the entire 'now' heap η_N garbage collected.

This idea of computing streams is made formal in the definition of the small-step operational semantics $\Longrightarrow$ for streams given in the left half of Figure 6. It starts by evaluating the term to a value of the form $v :: l$, which additionally produces the store $\sharp\eta_N\checkmark\eta_L$. Then the computation can be continued by evaluating $\mathsf{adv}\, l$ in the garbage collected store $\sharp\eta_L\checkmark$, which in turn produces a value $v' :: l'$ and a store $\sharp\eta'_N\checkmark\eta'_L$ – and so on.

Given a closed term t of type $\Box\mathsf{Str}(A)$, we compute the elements $v_1, v_2, v_3, \ldots$ of the stream defined by t as follows:

$$\langle \mathsf{unbox}\, t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3}{\Longrightarrow} \ldots$$

where we start with the empty heap $\emptyset$. Each state of the computation $\langle t_i; \eta_i\rangle$ consists of a term t_i and its 'now' heap η_i.

For example, consider the stream $\vdash \mathit{nats} : \Box(\mathsf{Str}(\mathsf{Nat}))$ defined in Section 2.2. To understand how this stream is executed, it is helpful to see how the definition of *nats* desugars to our core calculus. Namely, *nats* is defined as the term $\mathit{from} \boxdot \overline{0}$ where

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

Here we also unfold the definition of $\odot$. The first three steps of executing the *nats* stream look as follows:

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

As expected, the computation produces the consecutive natural numbers. In each step of the computation, the location l_i stores the fixed point *from* that underlies *nats*, whereas l_i' stores the computation that calls that fixed point with the current state of the computation, namely the number $\overline{i}$.

Our main result is that execution of programs by the machine in Figure 5 and Figure 6 is safe. For the stream semantics, this means that we can compute the stream defined by a term t of type $\Box(\mathsf{Str}(A))$ by successive unfolding ad infinitum as follows:

$$\langle \mathsf{unbox}\ t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3}{\Longrightarrow} \dots$$

This intuition is expressed more formally in the following theorem:

Theorem 3.1 (productivity). *Let A be a value type, i.e., a type constructed from $1, \mathsf{Nat}, +, \times$ only, and $\vdash t : \Box(\mathsf{Str}(A))$. Given any $n \in \mathbb{N}$, there is a reduction sequence*

$$\langle \mathsf{unbox}\ t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \quad \dots \quad \overset{v_n}{\Longrightarrow} \langle t_n; \eta_n\rangle \quad \textit{such that} \vdash v_i : A \textit{ for all } 1 \le i \le n.$$

3.3 Stream Transducer Semantics

More importantly, our language also facilitates stream processing, that is executing programs of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$. The small-step operational semantics $\overset{\cdot/\cdot}{\Longrightarrow}$ for executing such programs is given on the right half of Figure 6. So far the store has only been used by the term semantics to store delayed computations. In addition to that purpose, the stream transducer semantics uses the store to transfer the data received from the input steam to the stream transducer. To this end, we assume an arbitrary but fixed heap location l^*, which the machine uses to successively insert the input stream of type $\mathsf{Str}(A)$ as it becomes available. Note that the stream transducer semantics reserves the heap location l^* in new 'later' heap by storing $\langle\rangle$ in it. That means, l^* cannot be allocated by the machine and is available later when the input becomes available and needs to be stored in l^*.

Given a closed term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, we can execute it as follows:

$$\langle \mathsf{unbox}\ t\ (\mathsf{adv}\, l^*); \emptyset\rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2/v_2'}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3/v_3'}{\Longrightarrow} \dots$$

The machine starts with an empty heap $\emptyset$. In each step $\langle t_i; \eta_i\rangle \overset{v_{i+1}/v_{i+1}'}{\Longrightarrow} \langle t_{i+1}; \eta_{i+1}\rangle$, the machine starts in a state $\langle t_i; \eta_i\rangle$ consisting of a term t_i and heap η_i. Then it reads an input v_{i+1} and subsequently produces the output v_{i+1}' and the next state $\langle t_{i+1}; \eta_{i+1}\rangle$.

Let's consider a simple stream transducer to illustrate the workings of the semantics. The stream transducer *sum* takes a stream of numbers and computes at each point in time the sum of all previous numbers from the input stream. To this end *sum* uses the auxiliary function *sum′* that takes as additional argument the accumulator of type Nat.

$$\begin{aligned}&\mathit{sum}' : \Box\ (\mathsf{Nat} \to \mathsf{Str}\ (\mathsf{Nat}) \to \mathsf{Str}\ (\mathsf{Nat}))\\ &\mathit{sum}'\ \sharp\ \mathit{acc}\ (n :: \mathit{ns}) = (\mathit{acc} + n) :: \mathit{sum}' \odot (\mathit{acc} + n) \circledast \mathit{ns}\end{aligned}$$

$$sum : \Box\,(\mathsf{Str}\,(\mathsf{Nat}) \to \mathsf{Str}\,(\mathsf{Nat}))$$
$$sum = sum' \boxdot 0$$

To appreciate the workings of the stream transducer semantics, we desugar the definition of sum' in the surface syntax to our core calculus. In addition, we also unfold the definition of $\circledast$:

$$sum' = \mathsf{fix} f.\lambda acc.\lambda s.(acc + \mathsf{head}\, s) :: \mathsf{delay}\,(\mathsf{adv}\,(f \odot (acc + \mathsf{head}\, s))\,(\mathsf{adv}\,(\mathsf{tail}\, s)))$$

Let's look at the first three steps of executing the *sum* stream transducer. To this end, we feed the computation 2, 11, and 5 as input:

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

As expected, we receive 2, 13 (= 2 + 11), and 18 (= 2 + 11 + 5) as result. Moreover, in each step of the computation the location l_i stores the fixed point sum' that underlies the definition of *sum*, whereas l_i' stores the computation that calls that fixed point with the new accumulator value (0 + 2, 2 + 11, and 13 + 5, respectively) and the tail of the input stream.

Corresponding to the productivity property from the previous section, we prove the following causality property that states that the stream transducer semantics never gets stuck. To characterise the causality property, the theorem constructs a family of sets $T_k(A, B)$ which consists of states $\langle t; \eta \rangle$ on which the stream transducer machine can run for k more time steps.

Theorem 3.2 (causality). *Given any value types A and B, there is a family of sets $T_k(A, B)$ such that the following holds for all $k \in \mathbb{N}$:*

(i) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$ then $\langle \mathsf{unbox}\ t\ (\mathsf{adv}\, l^*); \emptyset \rangle \in T_k(A, B)$.
(ii) If $\langle t, \eta \rangle \in T_{k+1}(A, B)$ and $\vdash v : A$ then there are t', η', and $\vdash v' : B$ such that

$$\langle t; \eta \rangle \overset{v/v'}{\Longrightarrow} \langle t'; \eta' \rangle \ \text{ and } \ \langle t'; \eta' \rangle \in T_k(A, B).$$

That is, any term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$ defines a causal stream function which is effectively computed by the machine:

$$\langle \mathsf{unbox}\ t; \emptyset \rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_1; \eta_1 \rangle \overset{v_2/v_2'}{\Longrightarrow} \langle t_2; \eta_2 \rangle \overset{v_3/v_3'}{\Longrightarrow} \ldots$$

Note that the stream transducer semantics also extends to stream transducers with multiple streams as inputs. This can be achieved by a combinator *split* of type $\Box(\mathsf{Str}(A \times B) \to \mathsf{Str}(A) \times \mathsf{Str}(B))$. Similarly, the semantics also extends to transducers that take an event as input by virtue of a combinator *first* of type $\Box(\mathsf{Str}(1 + A) \to \mathsf{Ev}(A))$. Conversely, transducers producing events instead of streams can be executed using a combinator of type $\Box(\mathsf{Ev}(A) \to \mathsf{Str}(1 + A))$.

We give the proof of Theorem 3.1 and Theorem 3.2 in Section 6. Both results follow from a more general result for the machine, which is formulated using a Kripke logical relation.

3.4 Counterexamples

To conclude this section we review some programs that are rejected by our type system and illustrate their operational behaviour.

Unbox under delay. Recall the alternative definition of the stream of consecutive natural numbers *leakyNats* that uses the *map* combinator. First consider the definition of *leakyNats* in our core calculus:

$$\mathit{leakyNats} = \mathsf{fix}\, s.\overline{0} :: \mathsf{delay}\,(\mathsf{unbox}\,(\mathit{map}\,(\mathsf{box}(\lambda x.x + \overline{1})))) \circledast s$$

Let's contrast the execution of *nats* that we have seen in Section 3.2 with the execution of *leakyNats*:

$$\begin{array}{ll}
 & \langle \mathsf{unbox}\ \mathit{leakyNats}; \emptyset \rangle \\
\xRightarrow{\overline{0}} & \left\langle \mathsf{adv}\, l_1'; \ l_1 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\ l_1' \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_1) \right\rangle \\
\xRightarrow{\overline{1}} & \left\langle \begin{array}{ll} \mathsf{adv}\, l_2^3; & l_2^0 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\ l_2^1 \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_2^0), \\ & l_2^2 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_2^3 \mapsto \mathsf{adv}\, l_2^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_2^1))) \end{array} \right\rangle \\
\xRightarrow{\overline{2}} & \left\langle \begin{array}{ll} & l_3^0 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\ l_3^1 \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_3^0), \\ \mathsf{adv}\, l_3^5; & l_3^2 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_3^3 \mapsto \mathsf{adv}\, l_3^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_3^1))) \\ & l_3^4 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_3^5 \mapsto \mathsf{adv}\, l_3^4\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{1} :: l_3^3))) \end{array} \right\rangle \\
\vdots &
\end{array}$$

where $\mathit{step} = \mathsf{fix}\, f.\lambda s.\mathsf{unbox}\,(\mathsf{box}\,\lambda n.n + \overline{1})\,(\mathsf{head}\, s) :: (f \circledast \mathsf{tail}\, s)$.

While our type system rejects the term *leakyNats*, a corresponding term is typable in Krishnaswami's calculus [Krishnaswami 2013] and manifests the same memory allocation behaviour as *leakyNats* in our machine.

Lambda abstraction under delay. Recall the definition of the stream *leaky* from Section 2.3. It introduces a lambda abstraction in a later judgement and is therefore rejected by our type system.

$$\begin{array}{ll}
 & \langle \mathsf{unbox}\ \mathit{leaky}; \emptyset \rangle \\
\xRightarrow{\mathsf{true}} & \left\langle \mathsf{adv}\, l_1'; \begin{array}{l} l_1 \mapsto \mathsf{unbox}\ \mathit{leaky}', \\ l_1' \mapsto \mathsf{adv}\,(\mathbf{if}\,(\lambda x.\mathsf{true})\,\langle\rangle\,\mathbf{then}\, l_1\,\mathbf{else}\, l_1)\,(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true})) \end{array} \right\rangle \\
\xRightarrow{\mathsf{true}} & \left\langle \mathsf{adv}\, l_2'; \begin{array}{ll} & l_2 \mapsto \mathsf{unbox}\ \mathit{leaky}', \\ l_2' \mapsto & \mathsf{adv}\,(\mathbf{if}\,(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true}))\,\langle\rangle\,\mathbf{then}\, l_2\,\mathbf{else}\, l_2) \\ & \quad (\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_2\,\lambda y.\mathsf{true})) \end{array} \right\rangle \\
\not\Longrightarrow &
\end{array}$$

Note that the term $(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true}))$ from the heap after the first step is a value and thus appears unevaluated also in the heap after the second step. However, this term contains a reference to the heap location l_1, which has been garbage collected completing the second step. The machine thus ends up in a stuck state when it tries to dereference the garbage collected heap location l_1 during the third step.

4 GENERIC FRP LIBRARY

This section gives a number of higher-order FRP combinators in Simply RaTT, reminiscent of those found in libraries such as Yampa [Nilsson et al. 2002]. These can be used for programming with streams and events.

Perhaps the simplest example of a stream function is the constant stream over some element. Since we need to output this element in each time step in the future, we require it to come from a stable type. Thus the argument is of type $\Box A$.

$$\mathit{const} : \Box\, A \to \Box\, (\mathsf{Str}\ A)$$
$$\mathit{const}\ a\, \sharp\ = \mathsf{unbox}\ a :: \mathit{const}\ a$$

We can now recreate the *zeros* stream presented above as:

$$\mathit{zeros} : \mathsf{Str}\ (\mathsf{Nat})$$
$$\mathit{zeros} = \mathit{const}\ (\mathsf{box}\ 0)$$

Another simple way to generate a stream is to iterate a function $f : A \to A$ over some initial input, such that the output stream will be $(a, fa, f(fa), \ldots)$. Since we will keep using the function at every time step, it needs to be stable, i.e., $f : \Box(A \to A)$. Moreover, since we will keep a state with the current value of type A, that type A must be stable as well. We adopt a syntax like the one used in Haskell for type classes, to denote the additional requirement that a type is stable:

$$\mathit{iter} : A\ \mathsf{stable} \Rightarrow \Box\, (A \to A) \to \Box\, (A \to \mathsf{Str}\ A)$$
$$\mathit{iter}\ f\, \sharp\, \mathit{acc} = \mathit{acc} :: \mathit{iter}\ f \circledcirc (\mathsf{unbox}\ f\ \mathit{acc})$$

With this, we can define the stream of natural numbers:

$$\mathit{nats} : \Box\, (\mathsf{Str}\ \mathsf{Nat})$$
$$\mathit{nats} = \mathit{iter}\ (\mathsf{box}\ (\lambda n\,.\, n + 1)) \boxdot 0$$

We may also define a more general *iter* where A need not be stable:

$$\mathit{iter} : \Box\, (A \to \bigcirc\, A) \to \Box\, (A \to \mathsf{Str}\ A)$$
$$\mathit{iter}\ f\, \sharp\, \mathit{acc} = \mathit{acc} :: \mathit{iter}\ f \circledast (\mathsf{unbox}\ f\ \mathit{acc})$$

Given some stream, a standard operation in an FRP setting is to *filter* it according to some predicate. This behaviour is easy to implement in Simply RaTT, but because productivity forces us to output a value at each time step, if we take as input $\mathsf{Str}(A)$, we will need to output $\mathsf{Str}(\mathsf{Maybe}(A))$, where $\mathsf{Maybe}(A)$ is a shorthand for $1 + A$. Accordingly, we use the notation nothing and just t to denote $\mathsf{in}_1\, \langle\rangle$ and $\mathsf{in}_2\, t$, respectively.

$$\mathit{filter} : \Box\, (A \to \mathsf{Bool}) \to \Box\, (\mathsf{Str}\ A \to \mathsf{Str}\ (\mathsf{Maybe}\ A))$$
$$\mathit{filter}\ p = \mathit{map}\ \mathsf{box}\ (\lambda a\,.\, \textbf{if}\ \mathsf{unbox}\ p\ a\ \textbf{then}\ \mathsf{just}\ a\ \textbf{else}\ \mathsf{nothing})$$

To go from $\mathsf{Str}(\mathsf{Maybe}(A))$ and back to $\mathsf{Str}(A)$, we can use the *fromMaybe* function that replaces each missing value with a default value:

$$\mathit{fromMaybe} : \Box\, A \to \Box\, (\mathsf{Str}\ (\mathsf{Maybe}\ A) \to \mathsf{Str}\ A)$$
$$\mathit{fromMaybe}\ \mathit{def}\, \sharp\, (\mathsf{just}\ a \quad :: \mathit{as}) = a \qquad\qquad :: \mathit{fromMaybe}\ \mathit{def} \circledast \mathit{as}$$
$$\mathit{fromMaybe}\ \mathit{def}\, \sharp\, (\mathsf{nothing} :: \mathit{as}) = \mathsf{unbox}\ \mathit{def} :: \mathit{fromMaybe}\ \mathit{def} \circledast \mathit{as}$$

Given two streams, we can construct the product stream by simply "zipping" the two streams together. It is often easier to construct the more general version where a function is applied to each pair of inputs

$$\mathit{zipWith} : \Box\, (A \to B \to C) \to \Box\, (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ C)$$
$$\mathit{zipWith}\ f\, \sharp\, (a :: \mathit{as})\ (b :: \mathit{bs}) = \mathsf{unbox}\ f\ a\ b :: \mathit{zipWith}\ f \circledast \mathit{as} \circledast \mathit{bs}$$

The regular zip function is then defined as

$$\mathit{zip} : \Box\, (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A \times B))$$
$$\mathit{zip} = \mathit{zipWith}\ (\mathsf{box}\ (\lambda a\,.\, \lambda b\,.\, (a, b)))$$

Many applications require the ability to dynamically change the dataflow graph, e.g., when opening and closing windows in a GUI. Such behaviour can be implemented using *switches*, such

as the following, which given an initial stream and a stream event, outputs a stream following the initial stream until it receives a new one on its second argument

$$
\begin{aligned}
&\mathit{switch} : \Box\ (\mathsf{Str}\ A \to \mathsf{Ev}\ (\mathsf{Str}\ A) \to \mathsf{Str}\ A)\\
&\mathit{switch}\ \sharp\ (x :: xs)\ (\mathsf{wait}\ \mathit{fas}) = x :: \mathit{switch} \circledast xs \circledast \mathit{fas}\\
&\mathit{switch}\ \sharp\ xs \qquad (\mathsf{val}\ \mathit{ys}) \quad = \mathit{ys}
\end{aligned}
$$

As we have described above, we may define streams that require a state, but the state must be defined explicitly. An example is the *scan* function that given a binary operator and an initial state, will output the stream of successive application of the binary operator on the input stream

$$
\begin{aligned}
&\mathit{scan} : B\ \mathsf{stable} \Rightarrow \Box\ (B \to A \to B) \to \Box\ (B \to \mathsf{Str}\ A \to \mathsf{Str}\ B)\\
&\mathit{scan}\ f\ \sharp\ \mathit{acc}\ (a :: as) = \mathit{acc}' :: \mathit{scan}\ f \odot \mathit{acc}' \circledast as\\
&\quad \textbf{where}\ \mathit{acc}' = \mathsf{unbox}\ f\ \mathit{acc}\ a
\end{aligned}
$$

We can now redefine the *sum* function from Section 3.3 as follows:

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

In general, we can encode any computable stream in our language by virtue of the following *unfolding* combinator:

$$
\begin{aligned}
&\mathit{unfold} : \Box\ (X \to A \times \bigcirc\ X) \to \Box\ (X \to \mathsf{Str}\ A)\\
&\mathit{unfold}\ f\ \sharp\ x = \pi_1\ (\mathsf{unbox}\ f\ x) :: \mathit{unfold}\ f \circledast (\pi_2\ (\mathsf{unbox}\ f\ x))
\end{aligned}
$$

To further showcase programming with events, we define the function *await*, which listens for two events and produces a pair event that triggers after both events have arrived. As with *scan*, we need a state to keep the value of the first arriving event while waiting for the second one. This behaviour is implemented by two helper functions, which differ only in which element of the pair is given, and we only show one:

$$
\begin{aligned}
&\mathit{awaitA} : A\ \mathsf{stable} \Rightarrow \Box\ (A \to \mathsf{Ev}\ B \to \mathsf{Ev}\ (A \times B))\\
&\mathit{awaitA}\ \sharp\ a\ (\mathsf{wait}\ \mathit{eb}) = \mathsf{wait}\ (\mathit{awaitA} \odot a \circledast \mathit{eb})\\
&\mathit{awaitA}\ \sharp\ a\ (\mathsf{val}\ b) \quad = \mathsf{val}\ (a, b)
\end{aligned}
$$

We can now define *await* as

$$
\begin{aligned}
&\mathit{await} : A, B\ \mathsf{stable} \Rightarrow \Box\ (\mathsf{Ev}\ A \to \mathsf{Ev}\ B \to \mathsf{Ev}\ (A \times B))\\
&\mathit{await}\ \sharp\ (\mathsf{wait}\ \mathit{ea})\ (\mathsf{wait}\ \mathit{eb}) = \mathit{await} \circledast \mathit{ea} \circledast \mathit{eb}\\
&\mathit{await}\ \sharp\ (\mathsf{val}\ a) \quad \mathit{eb} \qquad = \mathsf{unbox}\ \mathit{awaitA}\ a\ \mathit{eb}\\
&\mathit{await}\ \sharp\ \mathit{ea} \qquad (\mathsf{val}\ b) \quad = \mathsf{unbox}\ \mathit{awaitB}\ b\ \mathit{ea}
\end{aligned}
$$

The requirement that A and B be stable is crucial since we need to keep the first arriving event until the second occurs.

A second example using events is the *accumulator* combinator. Given a value and a stream of events carrying functions, every time an event is received, the function is applied to the value and output as an event:

$$
\begin{aligned}
&\mathit{accum} : \Box\ A \to \Box\ (\mathsf{Str}\ (\mathsf{Ev}\ (A \to B)) \to \mathsf{Str}\ (\mathsf{Ev}\ B))\\
&\mathit{accum}\ a = \mathit{map}\ (\mathit{eventApp}\ a)
\end{aligned}
$$

The *accum* function uses the helper function below that takes a single event carrying a function and produces an event that applies the function to a given value:

$$\begin{aligned}&\mathit{eventApp} : \Box\, A \to \Box\, (\mathsf{Ev}\ (A \to B) \to \mathsf{Ev}\ B)\\ &\mathit{eventApp}\ a = \mathit{map}\ (\mathsf{box}\ (\lambda f\, .\, f\ (\mathsf{unbox}\ a)))\end{aligned}$$

5 SIMULATING LUSTRE

Lustre is a synchronous dataflow language. Programs describe dataflow graphs and evaluation proceeds in steps, reading input signals and producing output signals. Each signal is associated with a clock, which is always a sub-clock of the global clock, and as such can be described as a sequence of Booleans describing when the clock ticks. A pair of a clock and a signal that produces an output whenever the clock ticks is called a *flow*.

In Simply RaTT clocks can be encoded as Boolean streams and flows as streams of maybe values

$$\mathsf{Clock} = \mathsf{Str}(\mathsf{Bool}) \qquad\qquad \mathsf{Flow}(A) = \mathsf{Str}(\mathsf{Maybe}(A))$$

With these encodings, we now show how to encode some of the basic constructions of Lustre.

The clock associated to a flow ticks whenever the stream produces a value in A. For example, the *basic clock* of the system is the fastest possible clock and the clock *never* is the clock that never ticks. These can be defined as

$$\begin{aligned}&\mathit{basicClock} : \Box\, \mathsf{Clock} && \mathit{never} : \Box\, \mathsf{Clock}\\ &\mathit{basicClock} = \mathit{const}\ (\mathsf{box}\ \mathsf{true}) && \mathit{never} = \mathit{const}\ (\mathsf{box}\ \mathsf{false})\end{aligned}$$

In Simply RaTT, a cycle of the program corresponds to a single stream (transducer) unfolding.

Given a clock, we can slow it down to tick only at certain intervals. We define here a function that given a clock, slows it down to only tick every nth tick. Since we need to carry a state (how often to tick and what step we are at) we define first a helper function:

$$\begin{aligned}&\mathit{everyNthAux} : \mathsf{Nat} \to \Box\, (\mathsf{Nat} \to \mathsf{Clock} \to \mathsf{Clock})\\ &\mathit{everyNthAux}\ \mathit{step} \sharp \mathit{count}\ (c :: \mathit{cs}) = \mathbf{if}\ (\mathsf{unbox}\ (\mathsf{promote}\ \mathit{step}) = \mathit{count})\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ c \quad\ :: \mathit{everyNthAux}\ \mathit{step} \odot 0 \circledast \mathit{cs}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{false} :: \mathit{everyNthAux}\ \mathit{step} \odot (\mathit{count} + 1) \circledast \mathit{cs}\end{aligned}$$

We can now define the actual function by giving the helper function an initial state:

$$\begin{aligned}&\mathit{everyNth} : \mathsf{Nat} \to \Box\, (\mathsf{Clock} \to \mathsf{Clock})\\ &\mathit{everyNth}\ n = \mathit{everyNthAux}\ n \boxdot 0\end{aligned}$$

Given a flow and a clock, we can restrict the flow to that clock. If the clock is faster than the "internal" clock of the flow, this will not change the flow.

$$\begin{aligned}&\mathit{when} : \Box\, (\mathsf{Clock} \to \mathsf{Flow}\ A \to \mathsf{Flow}\ A)\\ &\mathit{when} \sharp (c :: \mathit{cs})\ (a :: \mathit{as}) = (\mathbf{if}\ c\ \mathbf{then}\ a\ \mathbf{else}\ \mathsf{nothing}) :: \mathit{when} \circledast \mathit{cs} \circledast \mathit{as}\end{aligned}$$

Recursive flows are implemented in Lustre using `pre` (previous) and `->` (followed by). We *could* implement these directly in Simply RaTT, but in many Lustre programs `pre` and `->` are used in a pattern that is very natural to Simply RaTT programs: `pre` is used to keep track of some state (that we may update) and `->` provides the initial state. As an example, consider the Lustre node that computes the flow of natural numbers:

```
n = 0 -> pre(n) + 1
```

The equivalent Simply RaTT program is

$$\begin{aligned}&\mathit{nats} : \Box\, (\mathsf{Flow}\ (\mathsf{Nat}))\\ &\mathit{nats} = \mathit{natsAux} \boxdot 0\end{aligned}$$

$$\begin{aligned}
&\textbf{where } \mathit{natsAux} : \Box\,(\mathsf{Nat} \to \mathsf{Flow}\,(\mathsf{Nat})) \\
&\qquad \mathit{natsAux} \sharp \mathit{pre} = \mathsf{just}\ \mathit{pre} :: \mathit{natsAux} \odot (\mathit{pre} + 1)
\end{aligned}$$

which is similarly composed of an initial state and then the actual computation, which may refer to the previous value.

As another example, consider the following Lustre node which takes a Boolean flow b as input:

```
edge = false -> (b and not pre(b))
```

This output flow is true when it detects a "rising edge" in its input flow, i.e., when the input goes from false to true. This is translated in a similar way to a helper function that does the computation:

$$\begin{aligned}
&\mathit{edgeAux} : \Box\,(\mathsf{Bool} \to \mathsf{Flow}\,(\mathsf{Bool}) \to \mathsf{Flow}\,(\mathsf{Bool})) \\
&\mathit{edgeAux} \sharp \mathit{pre}\ (\mathsf{just}\ b \quad :: bs) = b' \quad :: \mathit{edgeAux} \odot b' \circledast cs \\
&\mathit{edgeAux} \sharp \mathit{pre}\ (\mathsf{nothing} :: bs) = \mathit{pre} :: \mathit{edgeAux} \odot \mathit{pre} \circledast cs \\
&\quad \textbf{where } b' = (b\ \mathsf{and}\ (\neg\ \mathit{pre}))
\end{aligned}$$

and then a function giving the initial state:

$$\begin{aligned}
&\mathit{edge} : \Box\,(\mathsf{Flow}\,(\mathsf{Bool}) \to \mathsf{Flow}\,(\mathsf{Bool})) \\
&\mathit{edge} = \mathit{edgeAux} \boxdot \mathsf{false}
\end{aligned}$$

Since a flow is restricted to its internal clock, it may not produce anything at many ticks of the basic clock. To alleviate this, Lustre provides the `current` operator, which given a flow on a clock slower than the basic clock, fills the holes in the flow with whatever the latest values was. The equivalent Simply RaTT program is:

$$\begin{aligned}
&\mathit{current} : A\ \mathsf{stable} \Rightarrow \Box\,(\mathsf{Flow}\ A \to \mathsf{Flow}\ A) \\
&\mathit{current}\ as = \mathsf{box}\ (\lambda as\,.\,\mathsf{unbox}\ \mathit{currentAux}\ (\mathsf{head}\ as)\ as) \\
&\quad \textbf{where } \mathit{currentAux} : A\ \mathsf{stable} \Rightarrow \Box\,(\mathsf{Maybe}\ A \to \mathsf{Flow}\ A \to \mathsf{Flow}\ A) \\
&\qquad \mathit{currentAux} \sharp \mathit{pre}\ (\mathsf{just}\ a \quad :: as) = \mathsf{just}\ a :: \mathit{currentAux} \odot \mathsf{just}\ a \circledast as \\
&\qquad \mathit{currentAux} \sharp \mathit{pre}\ (\mathsf{nothing} :: as) = \mathit{pre} \quad :: \mathit{currentAux} \odot \mathit{pre} \circledast as
\end{aligned}$$

Our last example is an implementation of `counter` from the Lustre V6 manual [Erwan Jahier and Halbwachs 2019]. The `counter` program takes as input an initial value and a constant that determines how much to increment in each step. Its current state is stored in *pre*. It then listens to two flows, an "increment" flow and a "reset" flow. If the counter receives true on the increment flow, it increments the counter by the increment constant. If it receives true on the reset flow, it resets the counter to the initial value and otherwise it will continue in the same state.

$$\begin{aligned}
&\mathit{counter} : \mathsf{Nat} \to \mathsf{Nat} \to \Box\,(\mathsf{Nat} \to \mathsf{Flow\ Bool} \to \mathsf{Flow\ Bool} \to \mathsf{Flow\ Nat}) \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{just}\ s \quad :: ss)\ (\mathsf{just}\ r \quad :: rs) = \\
&\quad \textbf{if } s \textbf{ then } \mathsf{just}\ \mathit{init} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{init} \circledast ss \circledast rs \\
&\quad \textbf{else } \mathsf{just}\ (\mathit{pre} + \mathit{incr}) \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot (\mathit{pre} + \mathit{incr}) \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{nothing} :: ss)\ (\mathsf{just}\ r \quad :: rs) = \\
&\quad \textbf{if } r \textbf{ then } \mathsf{just}\ \mathit{init} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{init} \circledast ss \circledast rs \\
&\quad \textbf{else } \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{just}\ s \quad :: ss)\ (\mathsf{nothing} :: rs) = \\
&\quad \textbf{if } r \textbf{ then } \mathsf{just}\ (\mathit{pre} + \mathit{incr}) :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot (\mathit{pre} + \mathit{incr}) \circledast ss \circledast rs \\
&\quad \textbf{else } \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{nothing} :: ss)\ (\mathsf{nothing} :: rs) = \\
&\quad \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs
\end{aligned}$$

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{Nat}]\!]^{\overline{H}}_{\sigma} &= \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathcal{V}[\![1]\!]^{\overline{H}}_{\sigma} &= \{\langle\rangle\} \\
\mathcal{V}[\![A \times B]\!]^{\overline{H}}_{\sigma} &= \left\{ \langle v_1, v_2 \rangle \,\middle|\, v_1 \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \wedge v_2 \in \mathcal{V}[\![B]\!]^{\overline{H}}_{\sigma} \right\} \\
\mathcal{V}[\![A + B]\!]^{\overline{H}}_{\sigma} &= \left\{ \mathsf{in}_1\, v \,\middle|\, v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \right\} \cup \left\{ \mathsf{in}_2\, v \mid v \in \mathcal{V}[\![B]\!]^{\overline{H}}_{\sigma} \right\} \\
\mathcal{V}[\![A \to B]\!]^{\overline{H}}_{\sigma} &= \left\{ \lambda x.t \,\middle|\, \forall(\sigma' \diamond \overline{H}') \sqsupseteq (\mathrm{gc}(\sigma) \diamond \overline{H}). \forall w \in \mathcal{V}[\![A]\!]^{\overline{H}'}_{\sigma'}. t[w/x] \in \mathcal{T}[\![B]\!]^{\overline{H}'}_{\sigma'} \right\} \\
\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma} &= \left\{ v \,\middle|\, \forall \overline{H}' \leq_{\mathsf{suf}} \overline{H}. \mathsf{unbox}(v) \in \mathcal{T}[\![A]\!]^{\overline{H}'}_{\sharp} \right\} \\
\mathcal{V}[\![\bigcirc A]\!]^{()}_{\sigma} &= \{l \mid l \text{ is any heap location}\} \\
\mathcal{V}[\![\bigcirc A]\!]^{H;\overline{H}}_{\sigma} &= \left\{ l \,\middle|\, \forall \eta \in H. \sigma(l) \in \mathcal{T}[\![A]\!]^{\overline{H}}_{\mathrm{gc}(\sigma)\checkmark\eta} \right\} \\
\mathcal{V}[\![\mu\alpha.A]\!]^{\overline{H}}_{\sigma} &= \left\{ \mathsf{into}(v) \,\middle|\, v \in \mathcal{V}[\![A[\bigcirc(\mu\alpha.A)/\alpha]]\!]^{\overline{H}}_{\sigma} \right\}
\end{aligned}
$$

$$
\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma} = \left\{ t \,\middle|\, \forall(\sigma' \diamond \overline{H}') \sqsupseteq_{\checkmark} (\sigma \diamond \overline{H}). \exists \sigma'', v.\, \langle t; \sigma' \rangle \Downarrow \langle v; \sigma'' \rangle \wedge v \in \mathcal{V}[\![A]\!]^{\overline{H}'}_{\sigma''} \right\}
$$

$$
\begin{aligned}
C[\![\cdot]\!]^{\overline{H}}_{\perp} &= \{\star\} \\
C[\![\Gamma, x : A]\!]^{\overline{H}}_{\sigma} &= \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in C[\![\Gamma]\!]^{\overline{H}}_{\sigma}, v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \right\} \\
C[\![\Gamma, \checkmark]\!]^{\overline{H}}_{\sharp\eta_N\checkmark\eta_L} &= C[\![\Gamma]\!]^{\{\eta_L\},\overline{H}}_{\sharp\eta_N} \\
C[\![\Gamma, \sharp]\!]^{\overline{H}}_{\sigma} &= \bigcup_{\overline{H} \leq_{\mathsf{suf}} \overline{H}'} C[\![\Gamma]\!]^{\overline{H}'}_{\perp} \qquad \text{if } \sigma \neq \perp
\end{aligned}
$$

Garbage Collection:

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

Fig. 7. Logical Relation.

6 METATHEORY

Since the operational semantics rules out space leaks by construction, it only remains to be shown that the type system is sound, i.e., well-typed terms never get stuck. To this end, we devise a Kripke logical relation. Essentially, such a logical relation is a family $[\![A]\!]_w$ of sets of closed terms that satisfy the desired soundness property. This family of sets is indexed by w drawn from a suitable set of 'worlds' and is defined inductively on the structure of the type A and w. Then the proof of soundness is reduced to a proof that $\vdash t : A$ implies $t \in [\![A]\!]_w$ for all possible worlds. Finally we show how this soundness result is used to prove Theorem 3.1 and Theorem 3.2. All results have been formalised in the accompanying Coq proofs.

6.1 Worlds

To a first approximation, the worlds in our logical relation contain two components: a store σ and a number n, written $[\![A]\!]^n_\sigma$. The number index n allows us to define the logical relation for recursive types via step-indexing [Appel and McAllester 2001]. Concretely, this is achieved by defining $[\![\bigcirc A]\!]^{n+1}_\sigma$ in terms of $[\![A]\!]^n_\sigma$. Since unfolding recursive types $\mu\alpha.A$ to $A[\bigcirc\mu\alpha.A/\alpha]$ introduces a $\bigcirc$ modality, we thus achieve that the step index n decreases for recursive types. In essence, this means that terms in the logical relation $[\![A]\!]^n_\sigma$ can be executed safely for the next n time steps starting with

the store σ. Ultimately, the index σ enables us to prove that the garbage collection performed by the stream and stream transducer semantics (cf. Figure 6) is sound.

While this setup would be sufficient to prove soundness for the stream semantics (Theorem 3.1), it is not enough for the soundness of the stream transducer semantics (Theorem 3.2): To characterise our soundness property it is not enough to require that a term can be executed n more time steps. We also need to know what the input to a stream transducer of type $\mathrm{Str}(A) \to \mathrm{Str}(B)$ looks like, namely a stream where the first n elements are values of type A. We achieve this by describing what the heaps should look like in the next n time steps. Concretely, we assume a finite sequence $(H_1; \ldots; H_n)$, where each H_i is the set of heaps that we could potentially encounter i time steps into the future.

To summarise, the worlds in our logical relation consist of a store σ and a finite sequence $(H_1; \ldots; H_n)$ of sets of heaps. Instead of, $(H_1; \ldots; H_n)$ we also write $\overline{H}$, and we use the notation $(\sigma \diamond \overline{H})$ to refer to the world consisting of the store σ and the sequence $\overline{H}$. Intuitively, σ is the store for which the term in the logical relation can be safely executed, whereas each H_i in $\overline{H}$ contains all heaps for which the term can be safely executed after i time steps have passed.

A crucial ingredient of a Kripke logical relation is a preorder $\lesssim$ on the set of worlds such that the logical relation is closed under that preorder in the sense that $w \lesssim w'$ implies $[\![A]\!]_w \subseteq [\![A]\!]_{w'}$. To this end, we use a partial order $\sqsubseteq$ on heaps, which is the standard partial order on partial maps, i.e., $\eta \sqsubseteq \eta'$ iff $\eta(l) = \eta'(l)$ for all $l \in \mathrm{dom}\,(\eta)$. Moreover, we extend this order to stores in two different ways, resulting in the two orders $\sqsubseteq$ and $\sqsubseteq_\checkmark$:

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

That is, the heap order $\sqsubseteq$ is lifted to stores pointwise, whereas $\sqsubseteq_\checkmark$ extends $\sqsubseteq$ by defining $\sharp\eta_L \sqsubseteq_\checkmark \sharp\eta_N \checkmark \eta_L$. The more general order $\sqsubseteq_\checkmark$ is used in the logical relation, whereas the more restrictive $\sqsubseteq$ is needed to characterise the following property of the operational semantics:

Lemma 6.1. *Given any term t, value v, and pair of stores σ, σ' such that $\langle t; \sigma\rangle \Downarrow \langle v; \sigma'\rangle$, then $\sigma \sqsubseteq \sigma'$.*

We also extend the heap order $\sqsubseteq$ to sets of heaps and sequences of sets of heaps:

$$\begin{array}{rcll} H & \sqsubseteq & H' & \iff \forall \eta' \in H'. \exists \eta \in H. \eta \sqsubseteq \eta' \\ (H_1; H_2; \ldots; H_n) & \sqsubseteq & (H'_1; H'_2; \ldots; H'_n) & \iff \forall 1 \le i \le n. H_i \sqsubseteq H'_i \end{array}$$

That is, if $H \sqsubseteq H'$, then for every heap in H' there is a smaller one in H, and this ordering is lifted pointwise to finite sequences.

Finally, we combine these orderings to worlds pointwise as well

$$\begin{array}{rcl} (\sigma \diamond \overline{H}) & \sqsubseteq & (\sigma' \diamond \overline{H}') \iff \sigma \sqsubseteq \sigma' \quad \wedge \quad \overline{H} \sqsubseteq \overline{H}' \\ (\sigma \diamond \overline{H}) & \sqsubseteq_\checkmark & (\sigma' \diamond \overline{H}') \iff \sigma \sqsubseteq_\checkmark \sigma' \quad \wedge \quad \overline{H} \sqsubseteq \overline{H}' \end{array}$$

Note that whenever $\overline{H} \sqsubseteq \overline{H}'$, then $\overline{H}$ and $\overline{H}'$ are of the same length. That is, both $\overline{H}$ and $\overline{H}'$ describe the same number of future time steps. In order to describe a possible future world, i.e., after some time steps have passed, we use suffix ordering $\le_{\mathrm{suf}}$ on sequences. We say that $\overline{H}$ is a suffix of $\overline{H}'$, written $\overline{H} \le_{\mathrm{suf}} \overline{H}'$ iff there is a sequence $\overline{H}''$ such that H' is equal to the concatenation of $\overline{H}''$ and $\overline{H}$, written $\overline{H}''; \overline{H}$. Thus, a suffix $\overline{H} \le_{\mathrm{suf}} \overline{H}'$ describes a future state where the prefix $\overline{H}''$ has already been consumed.

6.2 Logical Relation

Our logical relation consists of two parts: A *value relation* $\mathcal{V}[\![A]\!]_w$ that contains all values that semantically inhabit type A at the world w, and a corresponding *term relation* $\mathcal{T}[\![A]\!]_w$ containing terms. Given a world $(\sigma \diamond \overline{H})$ we write $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ and $\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$ instead of $\mathcal{V}[\![A]\!]_{(\sigma \diamond \overline{H})}$ and $\mathcal{T}[\![A]\!]_{(\sigma \diamond \overline{H})}$, respectively. The two relations are defined by mutual induction in Figure 7. More precisely, the two relations are defined by well-founded induction by the lexicographic ordering on the triple $(|\overline{H}|, |A|, e)$, where $|\overline{H}|$ is the length of $\overline{H}$, $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$|\alpha| = |\bigcirc A| = |\mathsf{Nat}| = |1| = 1$$
$$|A \times B| = |A + B| = |A \to B| = 1 + |A| + |B|$$
$$|\Box A| = |\mu\alpha.A| = 1 + |A|$$

We define the size of $\bigcirc A$ to be the same as α. Thus $A[\bigcirc\mu\alpha.A/\alpha]$ is strictly smaller than $\mu\alpha.A$. This justifies the well-foundedness for recursive types. For types $\bigcirc A$, the well-foundedness of the definition can be observed by the fact that $\overline{H}$ is strictly shorter than $H;\overline{H}$, which is a shorthand notation for the sequence $(H; H_1; \ldots; H_n)$ where $\overline{H} = (H_1; \ldots; H_n)$.

The definition of the value relation for $\langle\rangle$, Nat, $\times$, and $+$ is standard. The definition of $\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma}$ expresses the fact that all its inhabitants can be evaluated safely at any time in the future. To express this, we use suffix ordering $\leq_{\mathsf{suf}}$. A value in $\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma}$ may be unboxed and subsequently evaluated at any time in the future, i.e., in the context of any suffix of $\overline{H}$.

The value relation for types $\bigcirc A$ encapsulates the soundness of garbage collection. The set $\mathcal{V}[\![\bigcirc A]\!]^{H;\overline{H}}_{\sigma}$ contains all heap locations that point to terms that can be executed safely in the next time step. The notation $\sigma(l)$ is a shorthand for $\eta_L(l)$ given that $\sigma = \sharp\eta_L$ or $\sigma = \sharp\eta_N \checkmark \eta_L$. Hence, we look up the location l in the 'later' heap of σ and require that the term that we find can be executed with the store obtained from σ by first garbage collecting the 'now' heap (if present) and extending it with any future heap drawn from H.

Garbage collection is also crucial in the definition of $\mathcal{V}[\![A \to B]\!]^{\overline{H}}_{\sigma}$, which only contains lambda abstractions that can be applied in a garbage collected store. This reflects the restriction of the typing rule for lambda abstraction, which requires the context Γ to be tick-free. The *leaky* example in Section 3.4 illustrates the necessity of this restriction. Semantically, this implies the following essential property of values:

Lemma 6.2. *For all $A, \sigma, \overline{H}$, we have that $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \subseteq \mathcal{V}[\![A]\!]^{\overline{H}}_{\mathrm{gc}(\sigma)}$.*

That is, after evaluating a term to a value, we can safely garbage collect the 'now' heap.

Finally, we obtain the soundness of the language by the following fundamental property of the logical relation $\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$.

Theorem 6.3 (Fundamental Property). *Given $\Gamma \vdash t : A$, and $\gamma \in C[\![\Gamma]\!]^{\overline{H}}_{\sigma}$, then $t\gamma \in \mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$.*

The theorem is proved by a lengthy but entirely standard induction on the typing relation $\Gamma \vdash t : A$. Two crucial ingredients to the proof are that all logical relations are closed under the ordering $\sqsubseteq_{\checkmark}$ on worlds, and that $C[\![\Gamma]\!]^{\overline{H}}_{\sigma}$ captures the correspondence between the tokens occurring in Γ and σ, namely they have the same number of locks and σ may not have fewer ticks than Γ.

6.3 Soundness of Stream and Stream Transducer Semantics

We conclude this section by demonstrating how we can use the fundamental property of our logical relation for proving the soundness of the abstract machines for evaluating streams (Theorem 3.1)

and stream transducers (Theorem 3.2), which amounts to proving productivity and causality of the calculus.

First, we observe that the operational semantics is deterministic:

Proposition 6.4 (deterministic machine).
(1) If $\langle t;\sigma\rangle \Downarrow \langle v_1;\sigma_1\rangle$ *and* $\langle t;\sigma\rangle \Downarrow \langle v_2;\sigma_2\rangle$, *then* $v_1 = v_2$ *and* $\sigma_1 = \sigma_2$.
(2) If $\langle t;\eta\rangle \overset{v_1}{\Longrightarrow} \langle t_1;\eta_1\rangle$ *and* $\langle t;\eta\rangle \overset{v_2}{\Longrightarrow} \langle t_2;\eta_2\rangle$, *then* $v_1 = v_2$, $t_1 = t_2$, *and* $\eta_1 = \eta_2$.
(3) If $\langle t;\eta\rangle \overset{v/v_1}{\Longrightarrow} \langle t_1;\eta_1\rangle$ *and* $\langle t;\eta\rangle \overset{v/v_2}{\Longrightarrow} \langle t_2;\eta_2\rangle$, *then* $v_1 = v_2$, $t_1 = t_2$, *and* $\eta_1 = \eta_2$.

Before we can prove Theorem 3.1, we need the following property of value types, i.e., types constructed from 1, Nat, +, ×

Lemma 6.5. *Let A be a value type and* $(\sigma \diamond \overline{H})$ *a world.*
(i) For all values v, we have that $v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ *iff* $\vdash v : A$.
(ii) $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ *is non-empty.*

Proof. By a straightforward induction on A. □

For the proof of Theorem 3.1, we construct for each type A the following family of sets $S_k(A)$, which intuitively contains all states on which the stream semantics can run for k more steps:

$$S_k(A) = \left\{ \langle t;\eta\rangle \;\middle|\; t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\eta\checkmark} \right\}$$

where $\{\emptyset\}$ is the singleton set containing the empty heap, and $\{\emptyset\}^k$ is the sequence containing k copies of $\{\emptyset\}$. Theorem 3.1 follows from the following lemma and the fact that the operational semantics is deterministic:

Lemma 6.6 (productivity). *Given any value type A, the following holds for all* $k \in \mathbb{N}$:
(i) If $\vdash t : \Box(\mathsf{Str}(A))$ *then* $\langle \mathsf{unbox}\, t;\emptyset\rangle \in S_k(A)$.
(ii) If $\langle t,\eta\rangle \in S_{k+1}(A)$ *then there are* t', η', *and* $\vdash v : A$ *such that*

$$\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle t';\eta'\rangle \;\text{ and }\; \langle t';\eta'\rangle \in S_k(A).$$

Proof.

(i) $\vdash t : \Box(\mathsf{Str}(A))$ implies $\sharp \vdash \mathsf{unbox}\, t : \mathsf{Str}(A)$ which by Theorem 6.3, implies that $\mathsf{unbox}\, t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp}$ and thus also $\mathsf{unbox}\, t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\checkmark}$. Hence $\langle \mathsf{unbox}\, t;\emptyset\rangle \in S_k(A)$.

(ii) Let $\langle t,\eta\rangle \in S_{k+1}(A)$. Then $t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sharp\eta\checkmark}$, which means that $\langle t;\sharp\eta\checkmark\rangle \Downarrow \langle w;\sigma\rangle$ and $w \in \mathcal{V}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$. Hence, $w = v :: l$ with $v \in \mathcal{V}[\![A]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$ and $l \in \mathcal{V}[\![\bigcirc\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$. Moreover, by Lemma 6.5 $\vdash v : A$ and by Lemma 6.1 $\sigma = \sharp\eta_N\checkmark\eta_L$. Hence, $\mathsf{adv}\, l \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\eta_L\checkmark}$. That is, $\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{adv}\, l;\eta_L\rangle$ and $\langle \mathsf{adv}\, l;\eta_L\rangle \in T_k(A)$. □

For the proof of causality we need the following property of the operational semantics, which essentially states that we never read from the 'later' heap.

Lemma 6.7. *If* $\langle t;\sigma, l \mapsto u\rangle \Downarrow \langle v;\sigma', l \mapsto u\rangle$, *then* $\langle t;\sigma, l \mapsto u'\rangle \Downarrow \langle v;\sigma', l \mapsto u'\rangle$ *for any* u'.

To prove the above lemma we have to make the following reasonable assumption about the function $\mathsf{alloc}\,(\cdot)$ that performs the allocation of fresh heap locations: Given two stores σ, σ' with $\mathsf{dom}\,(\mathsf{later}(\sigma)) = \mathsf{dom}\,(\mathsf{later}(\sigma'))$, we have that $\mathsf{alloc}\,(\sigma) = \mathsf{alloc}\,(\sigma')$. In other words, $\mathsf{alloc}\,(\sigma)$ only depends on the domain of the 'later' heap. For example, if the set of heap locations is just $\mathbb{N}$, then $\mathsf{alloc}\,(\sigma)$ could be implemented as the smallest heap location that is fresh for the 'later' heap of σ.

Analogously to the family of sets $S_k(A)$, we construct a family of sets $T_k(A,B)$ that contains all states which are safe to run for k more steps on the stream transducer semantics:

$$T_k(A,B) = \left\{ \langle t, \eta\rangle \;\middle|\; l^* \notin \operatorname{dom}(\eta) \wedge \forall v, w \in \mathcal{V}[\![A]\!]^{()}_{\bot}. t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^*} \right\}$$

where $H(A) = \left\{ l^* \mapsto v :: l^* \;\middle|\; v \in \mathcal{V}[\![A]\!]^{()}_{\bot} \right\}$ and $H^k(A)$ is the sequence of k copies of $H(A)$.

Finally, we give the proof of causality:

Proof of Theorem 3.2.

(i) Given $\vdash t : \Box(\mathrm{Str}(A) \to \mathrm{Str}(B))$ and $v, v' \in \mathcal{V}[\![A]\!]^{()}_{\bot}$, we need to show that $\mathrm{unbox}\, t\, (\mathrm{adv}\, l^*) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H(A)^k}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. By induction on $k+1$ we can show that $l^* \in \mathcal{V}[\![\bigcirc \mathrm{Str}(A)]\!]^{H^{k+1}(A)}_{\sharp l^* \mapsto v :: l^*}$. By definition of the value relation, this means that $v :: l^* \in \mathcal{T}[\![\mathrm{Str}(A)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$, which in turn implies that $\mathrm{adv}\, l^* \in \mathcal{T}[\![\mathrm{Str}(A)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. Since $\vdash t : \Box(\mathrm{Str}(A) \to \mathrm{Str}(B))$, we know that $\sharp \vdash \mathrm{unbox}\, t : \mathrm{Str}(A) \to \mathrm{Str}(B)$. Using Theorem 6.3 we thus obtain that $\mathrm{unbox}\, t \in \mathcal{T}[\![\mathrm{Str}(A) \to \mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp}$, which in turn implies that $\mathrm{unbox}\, t \in \mathcal{T}[\![\mathrm{Str}(A) \to \mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. Therefore, we have that $\mathrm{unbox}\, t\, (\mathrm{adv}\, l^*) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$.

(ii) Let $\langle t, \eta\rangle \in T_{k+1}(A,B)$ and $\vdash v : A$. We need to find l, η_N, η_L, and $\vdash v' : B$ such that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto \langle\rangle\rangle \Downarrow \langle v' :: l; \sharp\eta_N \checkmark \eta_L, l^* \mapsto \langle\rangle\rangle \tag{1}$$

$$\text{and} \quad \mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*} \quad \text{for all } w, w' \in \mathcal{V}[\![A]\!]^{()}_{\bot}. \tag{2}$$

By Lemma 6.5 (i), $v \in \mathcal{V}[\![A]\!]^{()}_{\bot}$, and by Lemma 6.5 (ii), there is some $w^* \in \mathcal{V}[\![A]\!]^{()}_{\bot}$. Since $t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w^* :: l^*}$, we have that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w^* :: l^*\rangle \Downarrow \langle v''; \sigma\rangle \text{ and } v'' \in \mathcal{V}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sigma}$$

Consequently, $v'' = v' :: l$ for some $v' \in \mathcal{V}[\![B]\!]^{H^{k+1}(A)}_{\sigma}$ by Lemma 6.1, σ is of the form $\sharp\eta_N \checkmark \eta_L, l^* \mapsto w^* :: l^*$. By Lemma 6.7 and Lemma 6.5 (i), we then have (1) and $\vdash v' : B$, respectively.

Finally, to prove (2), we assume $w, w' \in \mathcal{V}[\![A]\!]^{()}_{\bot}$ and show $\mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}$. Since $t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^*}$, we have that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^*\rangle \Downarrow \langle v'''; \sigma'\rangle \text{ and } v''' \in \mathcal{V}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sigma'}$$

By Lemma 6.7 and Proposition 6.4 we thus know that $v''' = v' :: l$ and $\sigma' = \sharp\eta_N \checkmark \eta_L, l^* \mapsto w :: l^*$. Consequently, $\sigma'(l) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}$, which implies that

$$\mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}.$$

□

7 RELATED WORK

The central ideas of functional reactive programming were originally developed for the language Fran [Elliott and Hudak 1997] for reactive animation. These ideas have since been developed into general purpose libraries for reactive programming, most prominently the Yampa library [Nilsson et al. 2002] for Haskell, which has been used in a variety of applications including games, robotics, vision, GUIs, and sound synthesis. Some of these libraries use a continuous notion of time, allowing

e.g., integrals over input data to be computed. A continuous notion of time can be encoded in Simply RaTT as well given that the language is extended with a type Time that suitably represents positive time intervals (e.g., floating-point numbers). For example, a Yampa-style signal function type from A to B is thus encoded as $\Box(\mathsf{Str}(\mathsf{Time}) \to \mathsf{Str}(A) \to \mathsf{Str}(B))$. This encoding reflects the (unoptimised) definition of Yampa-style signal functions [Nilsson et al. 2002], which is a coinductive type satisfying $\mathsf{SF}\ A\ B \cong \mathsf{Time} \to A \to (B \times \mathsf{SF}\ A\ B)$. We believe that it should be possible to implement a Yampa-style FRP library in Simply RaTT, and Section 4 has some examples of combinators similar to those found in Yampa. While some of these combinators have stability constraints on types, we believe that these constraints will always be satisfied in concrete applications.

Simply RaTT follows a *pull*-based approach to FRP, which means that the program is performing computations at every time step even if no event occurred. Elliott [2009] proposed an implementation of an FRP library that combines *pull*-based FRP with a *push*-based approach, where computation is only performed in response to incoming events. Whereas a pull-based approach is appropriate for example in games, which run at a fixed sampling rate, a push-based approach is more efficient for applications like GUIs, which often only need to react to events that occur infrequently.

The idea of using modal type operators for reactive programming goes back at least to the independent works of Jeffrey [2012]; Krishnaswami and Benton [2011] and Jeltsch [2013]. One of the inspirations for Jeffrey [2012] was to use linear temporal logic [Pnueli 1977] as a programming language through the Curry-Howard isomorphism. The work of Jeffrey and Jeltsch has mostly been based on denotational semantics, and Cave et al. [2014]; Krishnaswami [2013]; Krishnaswami and Benton [2011]; Krishnaswami et al. [2012] are the only works to our knowledge giving operational guarantees. The work of Cave et al. [2014] shows how one can encode notions of fairness in modal FRP, if one replaces the guarded fixed point operator with more standard (co)recursion for (co)inductive types.

The guarded recursive types and fixed point combinator originate with Nakano [2000], but have since been used for constructing logics for reasoning about advanced programming languages [Birkedal et al. 2011] using an abstract form of step-indexing [Appel and McAllester 2001]. The Fitch-style approach to modal types [Fitch 1952] has been used for guarded recursion in Clocked Type Theory [Bahr et al. 2017], where contexts can contain multiple, named ticks. Ticks can be used for reasoning about guarded recursive programs. The denotational semantics of Clocked Type Theory [Mannaa and Møgelberg 2018] reveals the difference from the more standard two-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. Combining these ideas with the garbage collection results of this paper, however, seems very difficult.

The previous work closest to the present work is that of Krishnaswami [2013]. We have already compared to this several times above, but give a short summary here. Simply RaTT is expressive enough to encompass all the positive examples of Krishnaswami's calculus, but we go a step further and identify a source of time leaks which allows us to eliminate in typing a number of leaking examples typable in Krishnaswami's calculus including the *leakyNats* example from the introduction, and *scary_const*. One might claim that these are explicit leaks, but detecting them in the type system is a major step forward we believe. Note that the Fitch-style approach is a real shift in approach: The time dependencies have changed, and Krishnaswami's context of stable variables has been replaced by a context of initial variables. One difference between these is that variables can be introduced from Krishnaswami's stable context. In Simply RaTT, initial variables

can generally not be introduced into temporal judgements. We plan to explain this change in terms of denotational semantics in future work.

Another approach to reactive programming is that of synchronous dataflow languages. Here the main abstraction is that of a "logical tick" or synchronous abstraction. This is the assumption that at each tick, the output is computed instantaneously from the input. This abstraction makes reasoning about time much easier than if we had to consider both the reactive behaviour and the internal timing behaviour of a program. Of particular interest is the synchronous dataflow language Lustre [Caspi et al. 1987]. Lustre is a first-order language used for describing and verifying real-time systems and is at the core of the SCADE industrial environment [Esterel Technologies SA 2019a] which is used for critical control systems in aerospace, rail transportation, industrial systems and nuclear power plants [Esterel Technologies SA 2019b]. In Section 5, we have shown how to encode some of the simpler concepts of Lustre in Simply RaTT, and how the concept of a logical tick fits well with the notion of stepwise stream unfolding.

8 CONCLUSIONS AND FUTURE WORK

We have presented the modal calculus Simply RaTT for reactive programming. Using the Fitch-style approach to modal types this gives a significant simplification of the type system and programming examples over existing approaches, in particular the calculus of Krishnaswami [2013]. Moreover, we have identified a source of time leaks and designed the type system to rule these out.

In future work we aim to extend Simply RaTT to a full type theory with dependent types for expressing properties of programs. Before doing that, however, we would like to extend Simply RaTT to encode fairness in types as in the work of Cave et al. [2014]. This is not easy, since it requires a distinction between inductive and coinductive guarded types, but Nakano's fixed point combinator forces these to coincide.

ACKNOWLEDGMENTS

This work was supported by a research grant (13156) from VILLUM FONDEN.

REFERENCES

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

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/LICS.2017.8005097

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

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

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

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

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

Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. *CoRR* abs/1804.05236 (2018), 1–21. arXiv:1804.05236 http://arxiv.org/abs/1804.05236

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.

258973

Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

Pascal Raymond Erwan Jahier and Nicolas Halbwachs. 2019. The LUSTRE V6 Reference Manual. https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf

Esterel Technologies SA. 2019a. Scientific Background. http://www.esterel-technologies.com/about-us/scientific-historic-background/.

Esterel Technologies SA. 2019b. Success Stories. http://www.esterel-technologies.com/success-stories/.

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

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

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

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

Wolfgang Jeltsch. 2013. Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

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

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

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

Bassel Mannaa and Rasmus Ejlers Møgelberg. 2018. The Clocks They Are Adjunctions: Denotational Semantics for Clocked Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs)*, Hélène Kirchner (Ed.), Vol. 108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, New York, NY, USA, 23:1–23:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.23

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, Napoli, IT.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

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

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

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

Marc Pouzet. 2006. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI* 1 (2006), 25.