Diamonds Are Not Forever

Liveness in Reactive Programming with Guarded Recursion

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

When designing languages for functional reactive programming (FRP) the main challenge is to provide the user with a simple, flexible interface for writing programs on a high level of abstraction while ensuring that all programs can be implemented efficiently in a low-level language. To meet this challenge, a new family of modal FRP languages has been proposed, in which variants of Nakano's guarded fixed point operator are used for writing recursive programs guaranteeing properties such as causality and productivity. As an apparent extension to this it has also been suggested to use Linear Temporal Logic (LTL) as a language for reactive programming through the Curry-Howard isomorphism, allowing properties such as termination, liveness and fairness to be encoded in types. However, these two ideas are in conflict with each other, since the fixed point operator introduces non-termination into the inductive types that are supposed to provide termination guarantees.

In this paper we show that by regarding the modal time step operator of LTL a submodality of the one used for guarded recursion (rather than equating them), one can obtain a modal type system capable of expressing liveness properties while retaining the power of the guarded fixed point operator. We introduce the language Lively RaTT, a modal FRP language with a guarded fixed point operator and an 'until' type constructor as in LTL, and show how to program with events and fair streams. Using a step-indexed Kripke logical relation we prove operational properties of Lively RaTT including productivity and causality as well as the termination and liveness properties expected of types from LTL. Finally, we prove that the type system of Lively RaTT guarantees the absence of implicit space leaks.

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, Linear Temporal Logic, Synchronous Data Flow Languages, Type Systems

ACM Reference Format:
Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds Are Not Forever: Liveness in Reactive Programming with Guarded Recursion. *Proc. ACM Program. Lang.* 5, POPL, Article 2 (January 2021), 28 pages. https://doi.org/10.1145/3434283

1 INTRODUCTION

Reactive programs such as servers and control software in cars, aircrafts and robots are traditionally written in imperative languages using a wide range of complex features including call-backs and shared state. For this reason, they are notoriously error-prone and hard to reason about. This

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.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/1-ART2
https://doi.org/10.1145/3434283

is unfortunate, since much of the most critical software currently in use is reactive. The goal of functional reactive programming (FRP) is to provide the programmer with tools for writing reactive programs on a high level of abstraction in the functional paradigm. In doing so, FRP extends the known benefits of functional programming also to reactive programming, in particular modularity and equational reasoning for programs. The challenge for achieving this goal is to ensure that all programs can be implemented efficiently in a low-level language.

From the outset, the central idea of FRP [Elliott and Hudak 1997] was that reactive programming simply is programming with signals and events. While elegant, this idea immediately leads to the question of what the interface for signals and events should be. A naive approach would be to model signals as streams in the sense of coinductive solutions to $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$, but this allows the programmer to write *non-causal* programs, i.e., programs where the present output depends on future input. Arrowised FRP [Nilsson et al. 2002], as implemented in the Yampa library for Haskell, solves this problem by taking signal functions as primitive rather than signals themselves. However, this approach forfeits some of the simplicity of the original FRP model and reduces its expressivity as it rules out useful types such as signals of signals.

More recently, a number of authors [Bahr et al. 2019; Jeffrey 2014; Jeltsch 2013; Krishnaswami 2013; Krishnaswami and Benton 2011; Krishnaswami et al. 2012] have suggested a modal approach to FRP in which causality is ensured through the introduction of a notion of time in the form of a modal operator. In this approach, an element of the modal type $\triangleright A$ should be thought of as data of type A arriving in the next time step. Signals should be modelled as a type of streams satisfying the type isomorphism $\mathsf{Str}(A) \cong A \times \triangleright\mathsf{Str}(A)$ capturing the idea that each pair of elements of a stream is separated by a time step. Events carrying data of type A can be represented by a type satisfying $\mathsf{Ev}(A) \cong A + \triangleright\mathsf{Ev}(A)$, stating that an event can either occur now, or at some point in the future. Types such as $\mathsf{Str}(A)$ and $\mathsf{Ev}(A)$ satisfying type equations in which the recursion variable is guarded by a $\triangleright$ are referred to as *guarded recursive types*. Combining this with guarded recursion [Nakano 2000] in the form of a fixed point operator of type $(\triangleright A \to A) \to A$ gives a powerful type system for reactive programming guaranteeing not only causality, but also productivity, i.e. the property that for a closed stream, each of its elements can always be computed in finite time. In some systems [Bahr et al. 2019; Krishnaswami 2013; Krishnaswami et al. 2012] the modal types have also been used to guarantee the lack of implicit space leaks, i.e., the problem of programs holding on to memory while continually allocating until they run out of space. These leaks have previously been a major problem in FRP.

Jeffrey [2012] suggested taking this idea further using Linear Temporal Logic (LTL) [Pnueli 1977] as a type system for FRP through the Curry-Howard isomorphism, a connection discovered independently by Jeltsch [2012]. This idea is not only conceptually appealing, but could also extend the expressivity of the type system considerably and have practical consequences. Indeed, LTL has a step modality $\bigcirc$ similar to $\triangleright$ used to express that a formula should be true one time step from now. It also has an operation $\square$ expressing global truth, i.e., formulas that hold now and at any time in the future. This operation has been used by Krishnaswami [2013] to express time-independent data that can be safely kept across time steps without causing space leaks. In this paper we are particularly interested in the *until* operator $\phi\ \mathcal{U}\ \psi$ of LTL, which expresses that ϕ holds now and for some more steps, after which ψ becomes true. Using this operator, we can encode the *finally* operator $\Diamond\phi$ as $\mathsf{tt}\ \mathcal{U}\ \phi$ stating that ϕ will eventually become true. In programming terms, the until operator is the inductive type given by constructors

$$\mathsf{now} : B \to A\ \mathcal{U}\ B \qquad\qquad \mathsf{wait} : A \to \bigcirc(A\ \mathcal{U}\ B) \to A\ \mathcal{U}\ B\,.$$

and the fact that it is inductive should imply a termination property similarly to that of LTL: Elements of type $A\ \mathcal{U}\ B$ will eventually produce a B after at most finitely many As, and similarly for elements of type $\Diamond B$. In programming, this can be used to express the property that a program will eventually produce an output, e.g., by timeout, or one can give a type of fair schedulers [Cave et al. 2014], see section 3.2 for details.

The goal of this paper is to define a language combining the expressive type system of LTL with the power of the guarded recursive fixed point combinator. Unfortunately, equating $\bigcirc$ and $\triangleright$ in such a system breaks the termination guarantee of the $\mathcal{U}$ type. For example, if $a : A$, the fixed point of $\mathsf{wait}\, a : \bigcirc(A\ \mathcal{U}\ B) \to A\ \mathcal{U}\ B$ will never produce a B. This is an example of a well-known phenomenon: The guarded fixed point combinator implies uniqueness of solutions to guarded recursive type equations like $X \cong B + A \times \triangleright X$, and so inductive and coinductive solutions coincide. In fact, the solutions behave more like coinductive types than inductive types and can even be used to encode coinductive types [Atkey and McBride 2013] in some settings.

This observation led Cave et al. [2014] to suggest removing the guarded recursive fixed point operator from FRP in order to distinguish between inductive and coinductive guarded types. This has the unfortunate effect of losing the power and elegance of the guarded fixed point operator for programming with coinductive types, which ought to be safe. Indeed it is well known that programming directly with coiteration is cumbersome and so most programming languages allow the programmer to construct elements of coinductive types using recursion. To guarantee productivity, one must use either the (non-modular) syntactic checks used in most proof assistants today, or sized types [Abel and Pientka 2013; Abel et al. 2017; Hughes et al. 1996; Sacchini 2013]. Given that the modal operator is in the language, guarded recursion is the most obvious solution to guaranteeing productivity.

1.1 Overview of Results

In this paper we show that by considering $\bigcirc$ a submodality of $\triangleright$, rather than equating them, we can use the guarded fixed point operator while retaining the termination guarantees of $\mathcal{U}$. Using $\triangleright$, the type $\mathsf{Ev}(A)$ of possibly occurring events of type A can be encoded as the unique solution to $\mathsf{Ev}(A) \cong A + \triangleright\mathsf{Ev}(A)$. Using $\bigcirc$, the type $\Diamond A$ of events of type A that must occur can be encoded as above. We will often refer to these as the types of possibly non-terminating and terminating events, respectively. The inclusion from $\bigcirc$ into $\triangleright$ can be used to type an inclusion of $\Diamond A$ into $\mathsf{Ev}(A)$. The lack of an inclusion from $\triangleright$ to $\bigcirc$ means that there is no inclusion $\triangleright\Diamond A \to \Diamond A$ to take a fixed point of to construct a diverging element of $\Diamond A$.

To make these ideas concrete we define the language *Lively RaTT* (section 2) as an extension of the language Simply RaTT [Bahr et al. 2019]. Simply RaTT is an FRP language with modal operators $\triangleright$ and $\Box$ as described above, as well as guarded recursive types and guarded fixed points. It uses a Fitch-style approach [Clouston 2018; Clouston et al. 2018; Fitch 1952] to programming with modal types, which means that the typing rules for introduction and elimination for modal types add and remove tokens from a context. This gives a direct style for programming with modalities, avoiding let-expressions as traditionally used for elimination. Lively RaTT has tokens $\checkmark_{\triangleright}$ and $\checkmark_{\bigcirc}$ for $\triangleright$ and $\bigcirc$, respectively, and the inclusion of $\bigcirc$ into $\triangleright$ is defined by allowing $\checkmark_{\triangleright}$ to eliminate also $\bigcirc$. We think of $\checkmark_{\bigcirc}$ and $\checkmark_{\triangleright}$ as a separation in time in judgements: Variables to the left of $\checkmark_{\bigcirc}$ or $\checkmark_{\triangleright}$ are available one time step before those to the right. The token $\checkmark_{\triangleright}$ is a stronger time step, allowing also recursive definitions to be unfolded. We illustrate the expressivity of Lively RaTT by showing how to program with events and fair streams in section 3.

We define two kinds of operational semantics for Lively RaTT (section 4): An evaluation semantics reducing terms to values at each time instant, and a step semantics capturing the dynamic behaviour of reactive programs over time. The latter is defined for streams, $\mathcal{U}$-types, and fair streams only. We prove causality and productivity of streams, and we prove the termination property for $\mathcal{U}$-types, i.e., that any term of type $A\ \mathcal{U}\ B$ eventually produces a B, also in a context of a stream of external

$$\begin{array}{rl}
\textbf{Types} & A, B ::= \alpha \mid 1 \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \Box A \mid \bigcirc A \mid \rhd A \mid \mathsf{Fix}\, \alpha.A \mid A\ \mathcal{U}\ B \\
\textbf{Stable types} & S, S' ::= 1 \mid \mathsf{Nat} \mid S \times S' \mid S + S' \mid \Box A \\
\textbf{Limit types} & L, L' ::= \alpha \mid 1 \mid \mathsf{Nat} \mid L \times L' \mid L + L' \mid A \to L \mid \Box L \mid \bigcirc L \mid \rhd A \mid \mathsf{Fix}\, \alpha.L
\end{array}$$

Fig. 1. Grammars for types, stable types and limit types. In typing rules, only closed types are considered.

$$\frac{}{\cdot \vdash} \qquad \frac{\Gamma \vdash \quad x \notin \mathrm{dom}\,(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \quad \mathsf{lock\text{-}free}(\Gamma)}{\Gamma, \sharp \vdash} \qquad \frac{\Gamma \vdash \quad \sharp \in \Gamma \quad m \in \{\bigcirc, \rhd\} \quad \mathsf{tick\text{-}free}(\Gamma)}{\Gamma, \checkmark_m \vdash}$$

Fig. 2. Well-formed contexts.

inputs. Using this, we prove that any term of the fair scheduler type can be unwound to a fair interleaving of streams, again also in a context of external input.

Finally, we show that the type system of Lively RaTT guarantees the lack of implicit space leaks. Our results on this extend those proved for Simply RaTT by Bahr et al. [2019] which in turn were based on a technique developed by [Krishnaswami 2013]. More precisely, our operational semantics stores input as well as delayed computations in a heap, and we show that it is safe to garbage collect the elements in the heap after two evaluation steps.

These results are proved (section 5) using an interpretation of types as sets of values indexed by four parameters, including an ordinal β. For finite β, this index should be thought of as a form of step-indexing: The interpretation of A at β in this case describes the behaviour of terms up to the first β evaluation steps. In our model, however, β runs all the way to $\omega \cdot 2$. The interpretation at higher β, in particular the limit ordinal ω describes global behaviour of programs.

The distinction between $\rhd$ and $\bigcirc$ can be seen in the model. At successor ordinals $\beta + 1$, the interpretation of $\rhd A$ and $\bigcirc A$ are both defined in terms of the interpretation of A at β in a step-indexed fashion [Birkedal et al. 2011], but at limit ordinals β, the interpretation of $\rhd A$ is the intersection of the interpretations at $\beta' < \beta$, whereas $\bigcirc A$ is interpreted using the interpretation of A at β. This interpretation of $\rhd A$ is needed to interpret fixed points, and the interpretation of $\bigcirc A$ ensures that the interpretation of $A\ \mathcal{U}\ B$ behaves globally as an inductive type.

The paper ends with an overview of related work (section 6) and conclusions, perspectives and future work (section 7). Full proofs can be found in the accompanying technical report.

2 LIVELY RATT

Lively RaTT is an extension of Simply RaTT [Bahr et al. 2019], a Fitch-style modal language for reactive programming. This section gives an overview of the language, referring to Figure 3 for an overview of the typing rules.

In the Fitch-style approach to modal types the introduction and elimination rules for these add and remove tokens from a context. For example, the modality $\bigcirc$ expresses delay of data by one time step and has introduction and elimination rules as follows (ignoring $\rhd$ for the moment).

$$\frac{\Gamma, \checkmark_{\bigcirc} \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : \bigcirc A} \qquad \frac{\Gamma \vdash t : \bigcirc A}{\Gamma, \checkmark_{\bigcirc}, \Gamma' \vdash \mathsf{adv}\, t : A}$$

The token $\checkmark_{\bigcirc}$ should be thought of as a separation by a single time step between the variables to the left of it and the rest of the judgement to the right. Thus the premise of the introduction rule

Simply typed λ-calculus:

$$\frac{\mathsf{token\text{-}free}(\Gamma') \vee A \text{ stable}}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \mathsf{tick\text{-}free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \pi_i\, t : A_i}$$

$$\frac{\Gamma \vdash t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2} \qquad \frac{\Gamma, x : A_i \vdash t_i : B \qquad \Gamma \vdash t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, t : \mathsf{Nat}} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma, x : \mathsf{Nat}, y : A \vdash t : A \qquad \Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, n) : A}$$

Modalities, $\mathcal{U}$-types, guarded recursion:

$$\frac{\Gamma, \surd_m \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : m\, A} \qquad \frac{\Gamma \vdash t : m\, A \qquad m \leqslant m' \vee A \text{ limit}}{\Gamma, \surd_{m'}, \Gamma' \vdash \mathsf{adv}\, t : A} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A}$$

$$\frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{now}\, t : A\, \mathcal{U}\, B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : \bigcirc(A\, \mathcal{U}\, B)}{\Gamma \vdash \mathsf{wait}\, s\, t : A\, \mathcal{U}\, B}$$

$$\frac{\Gamma, \sharp, x : B \vdash s : C \qquad \Gamma, \sharp, x : A, y : \bigcirc(A\, \mathcal{U}\, B), z : \bigcirc C \vdash t : C \qquad \Gamma, \sharp, \Gamma' \vdash u : A\, \mathcal{U}\, B}{\Gamma, \sharp, \Gamma' \vdash \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, u) : C}$$

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

Fig. 3. Typing rules. Here m, m' ranges over the set $\{\bigcirc, \rhd\}$ of time modalities ordered by $\bigcirc \leqslant \rhd$. In all rules, all contexts are assumed well-formed.

states that t has type A one time step after Γ, and so $\mathsf{delay}\, t$ has type $\bigcirc A$ at the time of Γ. Similarly, in the conclusion of the elimination rule, one time step has passed since the premise, so at that time t can be advanced to give an element of type A. This gives a direct approach to programming with modalities, as opposed to the more standard let-expressions. For example, delayed application of functions can be typed as

$$\lambda f.\lambda x.\mathsf{delay}((\mathsf{adv}\, f)(\mathsf{adv}\, x)) : \bigcirc(A \to B) \to \bigcirc A \to \bigcirc B \qquad (1)$$

Unlike Simply RaTT, Lively RaTT has two modalities for time delays: $\bigcirc$ and $\rhd$. Both correspond to a time step in the execution of reactive programs, but in addition, $\rhd$ corresponds to a time step in the sense of guarded recursion. Consequently, the $\surd_{\rhd}$ token is stronger than $\surd_{\bigcirc}$: Both can be used to advance time, but $\surd_{\rhd}$ can also be used to unfold fixed points. We capture this extra strength in a reflexive ordering generated by $\bigcirc \leqslant \rhd$ on delay modalities, and allowing $\surd_{m'}$ to eliminate modality m if $m \leqslant m'$. This induces an inclusion

$$\mathit{embed} = \lambda x.\mathsf{delay}(\mathsf{adv}\, x) : \bigcirc A \to \rhd A \qquad (2)$$

for all A. In general there is no inclusion in the opposite direction, except for a class of special types which we refer to as limit types, defined in Figure 1. The terminology refers to the step indexed interpretation of types, see section 5.

The tokens $\checkmark_{\triangleright}$ and $\checkmark_{\bigcirc}$ are collectively referred to as *ticks* and the rules in Figure 2 stipulate that there may be at most one tick in a context. This means that programs can refer to data from the present and previous time step, but not from earlier time steps. This is a crucial restriction that rules out implicit space leaks, and similar restrictions can be found in many other modal languages for FRP [Bahr et al. 2019; Cave et al. 2014; Krishnaswami 2013].

The second kind of token in Lively RaTT is $\sharp$, which separates the context into static variables to the left of $\sharp$ and dynamic variables to the right. Static variables are time-independent whereas the dynamic ones can depend on reactive data available only in the current instant. This distinction is only made once, so there can be at most one $\sharp$ in a context. The notion of time step is relevant only for dynamic variables, and therefore tokens $\checkmark_{\bigcirc}$ and $\checkmark_{\triangleright}$ can only appear to the right of a $\sharp$. The rules for well-formed contexts can be found in Figure 2.

The token $\sharp$ is associated with the modality $\square$. Data of type $\square A$ should be thought of as stable data, i.e., data that does not depend on time-dependent dynamic data, and can thus be safely transported into the future without causing space leaks. This is reflected in the introduction rule for $\square$ which ensures that $\mathsf{box}\, t$ can not contain free dynamic variables (i.e. variables to the right of $\sharp$), and in the elimination rule allowing $\Gamma \vdash t : \square A$ to be eliminated in context $\Gamma, \sharp, \Gamma'$ also when Γ' contains a tick.

Stable types (Figure 1) are types whose values by nature cannot contain time-dependent data, and so can be used in any dynamic context. This is implemented in the language by allowing variables of stable types to be introduced also over tokens. Generalising this to all variables would lead to space leaks. Note that function types are not stable since closures can contain time-dependent data.

Guarded recursive types are types of the form $\mathsf{Fix}\ \alpha.A$ satisfying the type isomorphism $\mathsf{Fix}\ \alpha.A \cong A[\triangleright(\mathsf{Fix}\ \alpha.A)/\alpha]$. Note that there is no restriction on A, which can in principle contain also negative occurrences of α, although none of the examples presented in this paper have this. The basic FRP types of streams and events can be encoded as guarded recursive types

$$\mathsf{Str}(A) \overset{\text{def}}{=} \mathsf{Fix}\ \alpha.A \times \alpha \qquad\qquad \mathsf{Ev}(A) \overset{\text{def}}{=} \mathsf{Fix}\ \alpha.A + \alpha$$

The fixed point combinator as defined by Nakano [2000] is simply a term of type $(\triangleright A \to A) \to A$. In FRP a few adjustments must be made to that. First of all, a fixed point will be called repeatedly at different dynamic times. To avoid space leaks, fixed points should therefore not have free dynamic variables (although the recursion variable itself should be dynamic), and the type of the fixed point should be of the form $\square A$. In Simply RaTT, the typing rule for fixed points states that $\Gamma \vdash \mathsf{fix}\ x.t : \square A$ if $\Gamma, \sharp, x : \triangleright A \vdash t : A$. In Lively RaTT this is too restrictive, since it prohibits nesting of guarded fixed points and recursion over elements of the until-types $A\ \mathcal{U}\ B$. The premise of the rule is therefore $\Gamma, x : \square\triangleright A, \sharp \vdash t : A$, which gives a more general fixed point rule.

As an example, mapping of functions over streams can be defined using fixed points as

$$\begin{aligned}&\mathit{map} : \square(A \to B) \to \square(\mathsf{Str}\ A \to \mathsf{Str}\ B)\\ &\mathit{map} = \lambda f\,.\,\mathsf{fix}\ m\,.\,\lambda a :: \mathit{as}\,.\,(\mathsf{unbox}\ f)\ a :: \mathsf{delay}\ ((\mathsf{adv}\ (\mathsf{unbox}\ m))\ (\mathsf{adv}\ \mathit{as}))\end{aligned}$$

where :: refers to the infix constructor for streams, which in the example is also used for pattern matching. Note that the input function f has type $\square(A \to B)$ since it must be called at all futures.

Lively RaTT features two kinds of inductive types. The first is the natural numbers with essentially the standard typing rules for 0, suc and recursion. Note that these apply in any context Γ independent of which tokens are in Γ. The second is the until-type of LTL, to be thought of as the inductive solution to $A\ \mathcal{U}\ B \cong B + A \times \bigcirc(A\ \mathcal{U}\ B)$. As for the natural numbers, there is no restriction on the context for the introduction rules, but the elimination rule is by nature dynamic, since elimination of an element of $A \times \bigcirc(A \,\mathcal{U}\, B)$ should recurse one time step from now on the advanced element of type $A \,\mathcal{U}\, B$. To avoid space leaks, the recursors should be stable, i.e., not depend on dynamic data. Thus eliminating from $A \,\mathcal{U}\, B$ into a type C requires recursors of type $\square(B \to C)$ and $\square(A \to \bigcirc(A \,\mathcal{U}\, B) \to \bigcirc C \to C)$.

Finally, note that Lively RaTT is a higher-order functional programming language with the restriction that lambda abstraction is only allowed in contexts with no $\checkmark_{\bigcirc}$ and $\checkmark_{\rhd}$. This restriction is inherited from Simply RaTT where it is necessary to guarantee the lack of space leaks. As we shall see, this appears not to be a limitation in practice.

3 PROGRAMMING IN LIVELY RATT

This section gives a number of examples of programming in Lively RaTT. First, we give a series of examples of programming with events. Secondly, we show how to encode *fairness* and how to implement a fair scheduler.

3.1 Events and Diamond

As described in the introduction, events that *may* occur can be encoded in Lively RaTT as $\mathsf{Ev}\,A \overset{\text{def}}{=} \mathsf{Fix}\,\alpha.A + \alpha$. For example, the event that loops forever can be defined as

$$\begin{aligned}&\mathit{loopEvent} : \square\mathsf{Ev}\ A\\&\mathit{loopEvent} = \mathsf{fix}\ e\,.\,\mathsf{into}\ (\mathsf{in}_2\ (\mathsf{unbox}\ e))\end{aligned}$$

The type of events that *must* occur can be encoded as the diamond modality from LTL, namely $\Diamond(A) \overset{\text{def}}{=} 1 \,\mathcal{U}\, A$. Below we will use the following shorthand when working with Ev and $\Diamond$:

$$\begin{aligned}&\mathsf{now}_{\Diamond} : A \to \Diamond A && \mathsf{now}_{\mathsf{Ev}} : A \to \mathsf{Ev}\ A\\&\mathsf{now}_{\Diamond}\ a = \mathsf{now}\ a && \mathsf{now}_{\mathsf{Ev}}\ a = \mathsf{into}\ (\mathsf{in}_1\ a)\\&\mathsf{wait}_{\Diamond} : \bigcirc\Diamond A \to \Diamond A && \mathsf{wait}_{\mathsf{Ev}} : \rhd\mathsf{Ev}\ A \to \mathsf{Ev}\ A\\&\mathsf{wait}_{\Diamond}\ d = \mathsf{wait}\ \langle\rangle\ d && \mathsf{wait}_{\mathsf{Ev}}\ e = \mathsf{into}\ (\mathsf{in}_2\ e)\end{aligned}$$

Here the $\mathsf{now}_{\Diamond}$ and $\mathsf{now}_{\mathsf{Ev}}$ maps are like the *return* map from a monad. Both Ev and $\Diamond$ further admit a map reminiscent of the *bind* map of a monad. For Ev this is given by:

$$\begin{aligned}&\mathit{bind}_{\mathsf{Ev}} : \square(A \to \mathsf{Ev}\ B) \to \square(\mathsf{Ev}\ A \to \mathsf{Ev}\ B)\\&\mathit{bind}_{\mathsf{Ev}} = \lambda f\,.\,(\mathsf{fix}\ b\,.\,\lambda \mathit{eva}\,.\,\mathsf{case}\ \mathit{eva}\ \mathsf{of}\ \mathsf{now}_{\mathsf{Ev}}\ a\,.\,(\mathsf{unbox}\ f)\ a\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{wait}_{\mathsf{Ev}}\ e\,.\,\mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e))\end{aligned}$$

where $\circledast$ is the infix notation of the delayed function call as defined in Equation 1, which can be given the more general type $m_1(A \to B) \to m_2(A) \to m_3(B)$ for $m_i \in \{\bigcirc, \rhd\}$ with $m_1, m_2 \leqslant m_3$. To see that $\mathit{bind}_{\mathsf{Ev}}$ is well-typed, consider the two cases. In the first case $a : A$ and hence, the unboxed f can be applied immediately. In the second case $e : \rhd\mathsf{Ev}\,A$ and $b : \square\rhd(\mathsf{Ev}\,A \to \mathsf{Ev}\,B)$. It then follows that $\mathsf{unbox}\,b : \rhd(\mathsf{Ev}\,A \to \mathsf{Ev}\,B)$ and thus, by a delayed function application, $(\mathsf{unbox}\,b) \circledast e : \rhd\mathsf{Ev}\,B$. This is then wrapped in $\mathsf{wait}_{\mathsf{Ev}}$ to produce an element of $\mathsf{Ev}\,B$ as needed. Note the requirement for the map f to be stable. This is because it might be applied *in the future.*

For $\Diamond$, we define the map

$$\begin{aligned}&\mathit{bind}_{\Diamond} : \square(A \to \Diamond B) \to \square(\Diamond A \to \Diamond B)\\&\mathit{bind}_{\Diamond} = \lambda f\,.\,\mathsf{box}\ (\lambda \mathit{dia}\,.\,\mathsf{rec}_{\mathcal{U}}\ (a\,.\,(\mathsf{unbox}\ f)\ a, u\ w\ d\,.\,\mathsf{wait}_{\Diamond}\ d, \mathit{dia}))\end{aligned}$$

where again f must be stable. To see that $\mathit{bind}_{\Diamond}$ is well typed, consider the base and recursion case. In the base $a : A$ and hence, the unboxed f can be applied immediately. In the recursion case $u : 1, w : \bigcirc(A \,\mathcal{U}\, B)$ and $d : \bigcirc\Diamond B$, hence also $\mathsf{wait}_{\Diamond}\ d : \Diamond B$ as required.

We will also use sugared syntax for recursive definitions, writing e.g. the definition of $bind_{\mathsf{Ev}}$ as

$$\begin{aligned}
&bind_{\mathsf{Ev}} : \Box(A \to \mathsf{Ev}\ B) \to \Box(\mathsf{Ev}\ A \to \mathsf{Ev}\ B)\\
&bind_{\mathsf{Ev}}\ f \,\sharp\, (\mathsf{now}_{\mathsf{Ev}}\ a) = (\mathsf{unbox}\ f)\ a\\
&bind_{\mathsf{Ev}}\ f \,\sharp\, (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ (bind_{\mathsf{Ev}}\ f) \circledast e)
\end{aligned}$$

The $\sharp$ separates the variables into those received before and after fix, and since the two cases define $bind_{\mathsf{Ev}}\, f$ by guarded recursion, this should be considered an atomic subexpression with type $\Box\rhd(\mathsf{Ev}\, A \to \mathsf{Ev}\, B)$. Similarly, the definition of $bind_{\Diamond}$ can be written in the sugared syntax as

$$\begin{aligned}
&bind_{\Diamond} : \Box(A \to \Diamond B) \to \Box(\Diamond A \to \Diamond B)\\
&bind_{\Diamond}\ f = \mathsf{box}\ bind'_{\Diamond}\\
&\text{where } bind'_{\Diamond} : \Diamond A \to \Diamond B\\
&\qquad bind'_{\Diamond}\ (\mathsf{now}_{\Diamond}\ a) = (\mathsf{unbox}\ f)\ a\\
&\qquad bind'_{\Diamond}\ (\mathsf{wait}_{\Diamond}\ d) = \mathsf{wait}_{\Diamond}\ (bind'_{\Diamond}\ d)
\end{aligned}$$

Here, the two cases of the recursive definition of $bind'_{\Diamond}$ are written as pattern matching syntax. In the second case the subterm $bind'_{\Diamond}\ d$ represents the recursive call and should therefore be read as having type $\bigcirc\Diamond B$. To elaborate such definitions back into $\mathsf{rec}_{\mathcal{U}}$, replace calls such as $bind'_{\Diamond}\ d$ with a fresh variable that represents the call to the recursor. We chose to use the above style to make it clear when delayed arguments are used and how they are passed around. See Appendix A for an overview of the elaboration process from surface syntax to the core calculus.

Since $\Diamond$ represents events that must occur, and Ev represents more general, possibly occurring, events there is an inclusion from $\Diamond$ to Ev. Using the above syntax, this is by $\mathcal{U}$-recursion as

$$\begin{aligned}
&diaInclusion : \Box(\Diamond A \to \mathsf{Ev}\ A)\\
&diaInclusion = \mathsf{box}\ diaInclusion'\\
&\text{where } diaInclusion' : \Diamond A \to \mathsf{Ev}\ A\\
&\qquad diaInclusion'\ (\mathsf{now}_{\Diamond}\ a) = \mathsf{now}_{\mathsf{Ev}}\ a\\
&\qquad diaInclusion'\ (\mathsf{wait}_{\Diamond}\ d) = \mathsf{wait}_{\mathsf{Ev}}\ (embed\ (diaInclusion'\ d))
\end{aligned}$$

This map makes crucial use of the fact that $\bigcirc$ is a sub-modality of $\rhd$ in the call to $embed$, as defined in Equation 2.

A further consequence of the sub-modality relation is that non-terminating events "overrule" terminating events. Consider Ev containing a $\Diamond$:

$$\begin{aligned}
&diamondEvent : \Box(\mathsf{Ev}\Diamond A \to \mathsf{Ev}\ A)\\
&diamondEvent = bind_{\mathsf{Ev}}\ diaInclusion
\end{aligned}$$

The converse, a function with type $\Diamond \mathsf{Ev}\, A \to \Diamond\, A$, can not be written in the language, since the inner event may be non-terminating.

There is in general no inclusion from Ev into $\Diamond$ because of the requirement that elements of $\Diamond A$ terminate. One solution is to wrap the conversion in a timeout, which will handle the non-terminating case. We must then supply a natural number, representing how many time steps to wait, and let the conversion fail if we go beyond that. We define by natural number recursion

$$\begin{aligned}
&timeout : \mathsf{limit}\ A \Rightarrow \mathsf{Nat} \to \Box(\mathsf{Ev}\ A \to \Diamond(1 + A))\\
&timeout\ 0 \qquad\quad \sharp\, (\mathsf{now}_{\mathsf{Ev}}\ a) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_2\ a)\\
&timeout\ 0 \qquad\quad \sharp\, (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_1\ \langle\rangle)\\
&timeout\ (\mathsf{suc}\ n) \,\sharp\, (\mathsf{now}_{\mathsf{Ev}}\ a) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_2\ a)\\
&timeout\ (\mathsf{suc}\ n) \,\sharp\, (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{wait}_{\Diamond}\ (\mathsf{delay}\ ((\mathsf{unbox}\ (timeout\ n))\ (\mathsf{adv}\ e)))
\end{aligned}$$

Typing of the fourth case uses the requirement A limit: The delay corresponds to a $\bigcirc$-tick in the context, which in general can not be used to advance $e : \rhd \mathsf{Ev}\, A$. It can in this case since $A\,\mathsf{limit}$ implies $\rhd\mathsf{Ev}\, A\,\mathsf{limit}$.

Even though we can not give a general function $\Diamond \mathsf{Ev} A \to \Diamond A$, we can use the above *timeout* to give a function $\Diamond \mathsf{Ev} A \to \Diamond(1 + A)$.

$$\begin{array}{l} \mathit{eventDiamond} : \mathsf{limit}\ A \Rightarrow \mathsf{Nat} \to \Box(\Diamond \mathsf{Ev}\ A \to \Diamond(1 + A)) \\ \mathit{eventDiamond}\ n = \mathit{bind}_\Diamond\ (\mathit{timeout}\ n) \end{array}$$

In general we cannot join two events of type $\Diamond A$ and $\Diamond B$ to an event of type $\Diamond(A \times B)$, i.e., waiting for both events to occur and pairing up the result. Doing so for arbitrary types A and B may lead to space leaks: If the two events occur at different times, the result of whichever event occurs first would need to be buffered until the second one occurs. However, if A and B are stable, we can explicitly buffer the early event occurrence, which allows us to implement the join. The implementation relies on two auxiliary functions that differ only in which order their arguments are given. We give only one of them, implemented by $\mathcal{U}$-recursion:

$$\begin{array}{l} \mathit{joinAuxA} : A\ \mathsf{stable} \Rightarrow \Box(\Diamond B \to A \to \Diamond(A \times B)) \\ \mathit{joinAuxA} = \mathsf{box}\ \mathit{joinAuxA}' \\ \text{where}\ \mathit{joinAuxA}'\ (\mathsf{now}_\Diamond\ b)\ a\ = \mathsf{now}_\Diamond\ \langle a, b\rangle \\ \qquad\quad \mathit{joinAuxA}'\ (\mathsf{wait}_\Diamond\ d)\ a = \mathsf{wait}_\Diamond\ ((\mathit{joinAuxA}'\ d) \odot a) \end{array}$$

where $\odot$ is the infix function

$$\begin{array}{l} \_ \odot \_ : A\ \mathsf{stable} \Rightarrow m_1\ (A \to B) \to A \to m_2\ B \\ f \odot a = \mathsf{delay}\ ((\mathsf{adv}\ f)\ a) \end{array}$$

where $m_1, m_2 \in \{\bigcirc, \rhd\}$ and $m_1 \leqslant m_2$. With the above, we can now define the join by $\mathcal{U}$-recursion:

$$\begin{array}{l} \mathit{join} : A, B\ \mathsf{stable} \Rightarrow \Box(\Diamond A \to \Diamond B \to \Diamond(A \times B)) \\ \mathit{join} = \mathsf{box}\ \mathit{join}' \\ \text{where}\ \mathit{join}' : \Diamond A \to \Diamond B \to \Diamond(A \times B) \\ \qquad \mathit{join}'\ (\mathsf{now}_\Diamond\ a)\ (\mathsf{now}_\Diamond\ b)\ \ = \mathsf{now}_\Diamond\ (a, b) \\ \qquad \mathit{join}'\ (\mathsf{now}_\Diamond\ a)\ (\mathsf{wait}_\Diamond\ d)\ \ = (\mathsf{unbox}\ \mathit{joinAuxA})\ (\mathsf{wait}_\Diamond\ d)\ a \\ \qquad \mathit{join}'\ (\mathsf{wait}_\Diamond\ d)\ (\mathsf{now}_\Diamond\ b)\ \ = (\mathsf{unbox}\ \mathit{joinAuxB})\ (\mathsf{wait}_\Diamond\ d)\ b \\ \qquad \mathit{join}'\ (\mathsf{wait}_\Diamond\ d)\ (\mathsf{wait}_\Diamond\ d') = \mathsf{wait}_\Diamond\ ((\mathit{join}'\ d) \circledast d') \end{array}$$

We now give a function constructing elements of $\Diamond$ by buffering data for a given number of time steps. For this we need a type of *temporal natural numbers* that will serve as a means to count time:

$$\mathsf{Nat}_\bigcirc = \Diamond 1$$

This type can be thought of as natural numbers, where the successor operation requires one time step to compute. The zero and successor can be encoded as:

$$\begin{array}{c} 0_\bigcirc = \mathsf{now}_\Diamond \langle\rangle \\ \mathsf{suc}_\bigcirc\ n = \mathsf{wait}_\Diamond\ n \end{array}$$

Any temporal natural number can be imported into the future by means of $\mathcal{U}$-recursion.

$$\begin{array}{l} \mathit{import} : \mathsf{Nat}_\bigcirc \to \bigcirc \mathsf{Nat}_\bigcirc \\ \mathit{import}\ 0_\bigcirc \qquad\quad = \mathsf{delay}\ (0_\bigcirc) \\ \mathit{import}\ (\mathsf{suc}_\bigcirc\ n) = \mathsf{delay}\ (\mathsf{suc}_\bigcirc\ (\mathsf{adv}\ (\mathit{import}\ n))) \end{array}$$

Given a natural number, we can convert it into a temporal natural number by recursion on natural numbers:

$timer : \mathsf{Nat} \to \mathsf{Nat}_{\bigcirc}$
$timer\ 0 \qquad = 0_{\bigcirc}$
$timer\ (\mathsf{suc}\ n) = \mathsf{suc}_{\bigcirc}\ (import\ (timer\ n))$

Intuitively speaking, given a natural number *n*, $timer$ *n* is a timer with *n* ticks.

The buffer function takes a temporal natural number and requires A to be stable, for the input to be buffered.

$buffer : A\ \mathsf{stable} \Rightarrow \square(\mathsf{Nat}_{\bigcirc} \to A \to \Diamond A)$
$buffer = \mathsf{box}\ buffer'$
where $buffer' : \mathsf{Nat}_{\bigcirc} \to A \to \Diamond A$
$\qquad buffer'\ 0_{\bigcirc}\ a \qquad = \mathsf{now}_{\Diamond}\ a$
$\qquad buffer'\ (\mathsf{suc}_{\bigcirc}\ n)\ a = \mathsf{wait}_{\Diamond}\ ((buffer'\ n) \odot a)$

As a final example of working with $\Diamond$ we define a simple server. First off we define the type of servers as

$$\mathsf{Server} := \mathsf{Fix}\,\alpha.\alpha \times (\mathsf{Req} \to (\Diamond \mathsf{Resp} \times \alpha))$$

where Req and Resp are the types of requests and responses, respectively. In each step, a server can receive at most one request, which must eventually give a response. In either case, the server will return a new server in the next time step, with a possibly updated internal state.

With the above, we can define a simple server that given a string *s* and a number *n*, returns $\langle s, m\rangle$ after *n* time steps, where *m* is the number of requests received. We set $\mathsf{Req} := \mathsf{Nat} \times \mathsf{String}$ and $\mathsf{Resp} := \mathsf{String} \times \mathsf{Nat}$, and consider String to be stable. The server is defined by guarded recursion:

$rServer : \square(\mathsf{Nat} \to \mathsf{Server})$
$rServer\ \sharp\ m = \mathsf{into}\ \langle rServerFst, rServerSnd\rangle$
where $rServerFst : \triangleright\mathsf{Server}$
$\qquad rServerFst = (\mathsf{unbox}\ rServer) \odot m$
$\qquad rServerSnd : (\mathsf{Nat} \times \mathsf{String}) \to (\Diamond(\mathsf{String} \times \mathsf{Nat}) \times \triangleright\mathsf{Server})$
$\qquad rServerSnd\ \langle n, s\rangle = \langle(\mathsf{unbox}\ buffer)\ (timer\ n)\ \langle s, m\rangle, (\mathsf{unbox}\ rServer) \odot (\mathsf{suc}\ m)\rangle$

The server can be run and initialized with 0:

$rServerRun : \square\mathsf{Server}$
$rServerRun = \mathsf{box}\ ((\mathsf{unbox}\ rServer)\ 0)$

3.2 Fair Streams

A stream of type $\mathsf{Str}(A + B)$ will in each step produce either a value of type A or of B. For example, we can implement a scheduler that interleaves two streams in an alternating fashion, dropping every other element of either stream:

$altStr : \square(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B))$
$altStr\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{in}_2\ b :: (\mathsf{unbox}\ altStr' \circledast as \circledast bs)$

$altStr' : \square(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B))$
$altStr'\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{in}_1\ a :: (\mathsf{unbox}\ altStr \circledast as \circledast bs)$

$altStr$ and $altStr'$ are defined by mutual guarded recursion. The former starts with taking an element from the second stream whereas the latter starts with taking from the first stream. This mutual recursive syntax translates to a single guarded fixed point in the calculus via a standard tupling construction that constructs both functions simultaneously:

$$\begin{aligned}
&\mathit{altStrMut} : \Box((\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A+B)) \times (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A+B)))\\
&\mathit{altStrMut} = \mathsf{fix}\ r\,.\,\langle \lambda as\,.\,\lambda bs\,.\,\mathsf{in}_2\ (\mathsf{head}\ bs) :: (\pi_2^{\triangleright}\ (\mathsf{unbox}\ r) \circledast \mathsf{tail}\ as \circledast \mathsf{tail}\ bs),\\
&\qquad\qquad\qquad \lambda as\,.\,\lambda bs\,.\,\mathsf{in}_1\ (\mathsf{head}\ as) :: (\pi_1^{\triangleright}\ (\mathsf{unbox}\ r) \circledast \mathsf{tail}\ as \circledast \mathsf{tail}\ bs)\rangle
\end{aligned}$$

In the above definition, the variable r is of type

$$\Box\triangleright((\mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Str}(A+B)) \times (\mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Str}(A+B)))$$

We then use the projections π_i lifted to $\triangleright$ to access the desired component from $\mathsf{unbox}\, r$:

$$\begin{aligned}
\pi_i^{\triangleright} &: \triangleright(A_1 \times A_2) \to \triangleright A_i\\
\pi_i^{\triangleright} &= \lambda x.\mathsf{delay}(\pi_i(\mathsf{adv}\, x))
\end{aligned}$$

Given the above tupling construction, altStr is then defined as $\mathsf{box}(\pi_1\ (\mathsf{unbox}\ \mathit{altStrMut}))$. From now on we will use the mutual guarded recursion syntax with the understanding that it can be turned into a single guarded fixed point by tupling.

The following function inhabits the same type as altStr, but it only draws elements from the first stream, dropping the second stream altogether:

$$\begin{aligned}
&\mathit{dropSnd} : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A+B))\\
&\mathit{dropSnd} \sharp\ as\ bs = \mathsf{unbox}\ (\mathit{map}\ (\mathsf{box}\ \mathsf{in}_1))\ as
\end{aligned}$$

Following the work by Cave et al. [2014], we can refine the type $\mathsf{Str}(A+B)$ to a type $\mathsf{Fair(A, B)}$, whose inhabitants will produce in each step a value of type A or of type B, in a fair manner:

$$\mathsf{Fair(A,B)} = \mathsf{Fix}\,\alpha.A\ \mathcal{U}\ (B \times \triangleright(B\ \mathcal{U}\ (A \times \alpha)))$$

A term of type $\mathsf{Fair(A, B)}$ may first produce some elements of type A, but must after finitely many steps produce an element of type B. It may continue to produce more elements of type B, but must eventually produce an element of type A and then continue in this manner indefinitely. This required behaviour prevents us from implementing $\mathit{dropSnd}$ to produce a fair stream of type $\mathsf{Fair(A, B)}$. On the other hand, we can re-implement altStr to produce a fair stream as follows:

$$\begin{aligned}
&\mathit{altFair}\ : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B))\\
&\mathit{altFair}\ \sharp\,(a :: as)\ (b :: bs) = \mathsf{into}\ (\mathsf{now}\ \langle b, \mathsf{unbox}\ \mathit{altFair}' \circledast as \circledast bs\rangle)\\
&\mathit{altFair}' : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to B\ \ \mathcal{U}\ \ (A \times \triangleright\mathsf{Fair}(A,B)))\\
&\mathit{altFair}' \sharp\,(a :: as)\ (b :: bs) = (\mathsf{now}\ \langle a, \mathsf{unbox}\ \mathit{altFair} \circledast as \circledast bs\rangle)
\end{aligned}$$

In order to simplify programming with fair streams, we define shortcut constructors for the type $\mathsf{Fair(A, B)}$. To this end we define the following variant of the type $\mathsf{Fair(A, B)}$:

$$\mathsf{Fair'(B,A)} = B\ \mathcal{U}\ (A \times \triangleright\mathsf{Fair(A,B)})$$

We now have that $\mathsf{Fair(A, B)}$ unfolds to $A\ \mathcal{U}\ (B \times \triangleright\mathsf{Fair'(B,A)})$ and thus the two types $\mathsf{Fair(A, B)}$ and $\mathsf{Fair'(A, B)}$ are isomorphic. Fair streams are constructed by either staying with the first type A or switching to the second type B.

$$\begin{aligned}
&\mathsf{stay} : A \to \bigcirc\mathsf{Fair}(A,B) \to \mathsf{Fair}(A,B) && \mathsf{stay'} : A \to \bigcirc\mathsf{Fair'}(A,B) \to \mathsf{Fair'}(A,B)\\
&\mathsf{stay}\ a\ d\ \ = \mathsf{into}\ (\mathsf{wait}\ a\ d) && \mathsf{stay'}\ a\ d\ \ = \mathsf{wait}\ a\ d\\
&\mathsf{switch} : B \to \triangleright\mathsf{Fair'}(B,A) \to \mathsf{Fair}(A,B) && \mathsf{switch'} : B \to \triangleright\mathsf{Fair}(B,A) \to \mathsf{Fair'}(A,B)\\
&\mathsf{switch}\ b\ d = \mathsf{into}\ (\mathsf{now}\ \langle b, d\rangle) && \mathsf{switch'}\ b\ d = \mathsf{now}\ \langle b, d\rangle
\end{aligned}$$

From the types above one can immediately see that we can only stay with the same type finitely often – indicated by the $\bigcirc$ modality – whereas we can switch arbitrarily – indicated by the $\triangleright$ modality. Using the above shorthands, the $\mathit{altFair}$ function can thus be implemented more concisely as follows:

$$\begin{array}{l}
\mathit{altFair} \;: \square(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B)) \\
\mathit{altFair}\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{switch}\ b\ (\mathsf{unbox}\ \mathit{altFair}' \circledast as \circledast bs) \\
\mathit{altFair}' : \square(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}'(B,A)) \\
\mathit{altFair}'\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{switch}'\ a\ (\mathsf{unbox}\ \mathit{altFair} \circledast as \circledast bs)
\end{array}$$

The fair stream type Fair(A, B) can be considered a special case of the stream type $\mathsf{Str}(A + B)$ with additional liveness constraints. We can always forget these constraints by converting a fair stream into a normal stream:

$$\begin{array}{l}
\mathit{runFair} : \square(\mathsf{Fair}(A,B) \to \mathsf{Str}\ (A+B)) \\
\mathit{runFair}\ \sharp\ = \mathit{run}_1 \\
\quad \text{where } \mathit{run}_2 : \mathit{Fair}'\ (B,A) \to \mathsf{Str}\ (A+B) \\
\qquad\quad \mathit{run}_2\ (\mathsf{stay}'\ b\ d) \quad\; = \mathsf{in}_2\ b :: \mathit{embed}\ (\mathit{run}_2\ d) \\
\qquad\quad \mathit{run}_2\ (\mathsf{switch}'\ a\ d) = \mathsf{in}_1\ a :: \mathsf{unbox}\ \mathit{runFair} \circledast d \\
\qquad\quad \mathit{run}_1 : \mathsf{Fair}\ (A,B) \to \mathsf{Str}\ (A+B) \\
\qquad\quad \mathit{run}_1\ (\mathsf{stay}\ a\ d) \quad\; = \mathsf{in}_1\ a :: \mathit{embed}\ (\mathit{run}_1\ d) \\
\qquad\quad \mathit{run}_1\ (\mathsf{switch}\ b\ d) = \mathsf{in}_2\ b :: \mathsf{delay}\ (\mathit{run}_2\ (\mathsf{adv}\ d))
\end{array}$$

The function *runFair* is defined by guarded recursion with two nested $\mathcal{U}$-recursions on the two nested $\mathcal{U}$-types that make up the fair stream type. Note that the two recursive calls $\mathit{run}_1\, d$ and $\mathit{run}_2\, d$ produce a delayed stream of type $\bigcirc(\mathsf{Str}(A + B))$. Therefore, we have to use *embed* to convert them to type $\triangleright(\mathsf{Str}(A + B))$.

We conclude with an example that implements a more interesting interleaving of two streams into a fair stream, namely the fair scheduler from Cave et al. [2014] that selects a progressively increasing number of elements from the first stream for each time it selects an element from the second stream:

$$\begin{array}{l}
\mathit{sch} \;: \mathsf{limit}\ A, \mathsf{limit}\ B \Rightarrow \square(\mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B)) \\
\mathit{sch}\ \sharp\ n\ as\ bs = \mathit{until}\ (\mathit{timer}\ n)\ n\ as\ bs \\
\quad \text{where } \mathit{until} : \mathsf{Nat}_{\bigcirc} \to \mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B)) \\
\qquad\quad \mathit{until}\ (\mathsf{suc}_{\bigcirc}\ n)\ m\ (a :: as)\ (b :: bs) = \mathsf{stay}\ a\ (\mathit{until}\ n \odot m \circledast as \circledast bs)) \\
\qquad\quad \mathit{until}\ 0_{\bigcirc}\ m\ (a :: as)\ (b :: bs) \\
\qquad\qquad = \mathsf{switch}\ b\ (\mathsf{unbox}\ \mathit{sch}' \odot m + 1 \circledast as \circledast bs) \\
\mathit{sch}' : \mathsf{limit}\ A, \mathsf{limit}\ B \Rightarrow \square(\mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}'(B,A)) \\
\mathit{sch}'\ \sharp\ n\ (a :: as)\ (b :: bs) = \mathsf{switch}'\ a\ (\mathsf{unbox}\ \mathit{sch} \odot n \circledast as \circledast bs)
\end{array}$$

In particular unbox $\mathit{sch}\ 0\ as\ bs$ produces a fair stream of the following form:

$$B \quad A \quad A \quad B \quad A \quad A \quad A \quad B \quad A \quad A \quad A \quad A \quad B \quad A \quad A \quad A \quad A \quad A \quad B\ldots$$

The fair scheduler is implemented by guarded mutual recursion with a nested $\mathcal{U}$-recursion. The natural number is first turned into a timer, which is then recursed over using the *until* function. In each recursive step of *until* – corresponding to a tick of the timer – we select from the first stream. But once the timer reaches $0_{\bigcirc}$, we switch to selecting from the second stream, then immediately switch to selecting from the first stream again, increment the counter m, and proceed by guarded recursion.

Note that we require A and B to be limit types so that in turn $\mathsf{Str}(A)$ and $\mathsf{Str}(B)$ are limit types. The latter is needed in the first clause of the *until* function so that we may apply the recursive call $\mathit{until}\, n$ of type $\bigcirc(\mathsf{Nat} \to \mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Fair}(\mathsf{A}, \mathsf{B}))$ to both as and bs, which are of type $\triangleright\mathsf{Str}(A)$ and $\triangleright\mathsf{Str}(B)$, respectively.

4 OPERATIONAL SEMANTICS

The operational semantics of Lively RaTT is divided into two parts: an *evaluation semantics* that captures the computational behaviour at each time instant (section 4.1), and a *step semantics* that describes the dynamic behaviour of a Lively RaTT program over time. We introduce the latter in two stages. At first we only look at programs without external input (section 4.2). Afterwards we extend the semantics to account for programs that react to external inputs (section 4.3), e.g., terms of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, which continuously read inputs of type A and produce outputs of type B. Along the way we give a precise account of our main technical results, namely productivity, termination, liveness, and causality properties of the operational semantics, as well as the absence of implicit space leaks. To prove the latter, the evaluation semantics is formulated using a store on which external inputs and delayed computations are placed. At each reduction step, the step semantics garbage collects all elements of the store that are more than one step old, thereby avoiding implicit space leaks by construction.

4.1 Evaluation Semantics

The *evaluation semantics* is presented as a big-step operational semantics in Figure 4 and describes how a configuration consisting of a term t and a store σ evaluates to a value v and an updated store τ in the current time instant, denoted $\langle t; \sigma\rangle \Downarrow \langle v; \tau\rangle$. In the machine, unlike the surface language, terms can contain locations l, to be thought of as locations in the store. Formally, these range over a given set Loc of locations divided into a countably infinite collection of namespaces each consisting of countably infinitely many locations. The grammar below describes which terms of the calculus are considered values:

$$v, w ::= \langle\rangle \mid 0 \mid \mathsf{suc}\, v \mid \lambda x.t \mid \langle v, w\rangle \mid \mathsf{in}_i\, v \mid \ \mathsf{box}\, t \mid \mathsf{delay}\, t \mid \mathsf{fix}\, x.t \mid l \mid \mathsf{into}\, v \ \mid \mathsf{now}\, v \mid \mathsf{wait}\, v\, w$$

A store can be of one of three forms: $\sigma ::= \bullet \mid \eta_L \mid \eta_N \checkmark \eta_L$. The null store $\bullet$ is used for a special state of the machine in which it can neither write to the store, nor read from it. In the other two forms η_L (the 'later' heap) and η_N (the 'now' heap) are heaps, i.e., pairs of a namespace and a finite mapping from the namespace to terms. In either of the two latter cases, the evaluation semantics can update all heaps present by allocating fresh locations and placing delayed computations in them, but it can only read from the η_N heap. The notation $\sigma, l \mapsto t$ refers to an extension of the heap furthest to the right in σ, and $\mathsf{alloc}\,(-)$ is a function returning a fresh location in the heap furthest to the right.

The fragment of Lively RaTT consisting of the lambda calculus with sums, products, and natural numbers is given a standard call-by-value semantics. The non-standard parts of the semantics involve the three modalities $\Box$, $\bigcirc$, and $\triangleright$; the recursion principle for $\mathcal{U}$ types; and the fixed point combinator.

The constructors for the three modalities – box and delay – have a call-by-name semantics and produce suspended computations. Terms of the form $\mathsf{box}\, t$ are values consisting of unevaluated terms t, which are only evaluated once they are consumed by unbox. Terms of the form $\mathsf{delay}\, t$ are computations that may be executed in the next time step. These computations are suspended and placed on the heap, since the evaluation semantics only describes computations at the current time instant. In the next time instant, these will be executed if adv is applied to their location. The safety of the step semantics shows that such delayed computations will only be executed in the next time step, and do not need to be stored for future steps. Note that in some special cases, a term $\mathsf{delay}(t)$ will be executed immediately, for example when evaluating a closed term of the form $\mathsf{adv}(\mathsf{delay}(t))$. Although such a term is not well-typed as a closed term by itself, it can occur in the evaluation of the step semantics.

Call-by-value λ-calculus:

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

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

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

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

Modalities, $\mathcal{U}$-types, guarded recursion:

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

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

$$\frac{\langle n;\sigma\rangle \Downarrow \langle 0;\sigma'\rangle \qquad \langle s;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s, x\,y.t, n);\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle n;\sigma\rangle \Downarrow \langle \mathsf{suc}\, v;\sigma'\rangle \qquad \langle \mathsf{rec}_{\mathsf{Nat}}(s, x\,y.t, v);\sigma'\rangle \Downarrow \langle v';\sigma''\rangle \qquad \langle t[v/x, v'/y];\sigma''\rangle \Downarrow \langle w;\sigma'''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s, x\,y.t, n);\sigma\rangle \Downarrow \langle w;\sigma'''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{now}\, t;\sigma\rangle \Downarrow \langle \mathsf{now}\, v;\sigma'\rangle} \qquad \frac{\langle t_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \qquad \langle t_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle}{\langle \mathsf{wait}\, t_1\, t_2;\sigma\rangle \Downarrow \langle \mathsf{wait}\, v_1\, v_2;\sigma''\rangle}$$

$$\frac{\langle u;\sigma\rangle \Downarrow \langle \mathsf{now}\, v;\sigma'\rangle \qquad \langle s[v/x];\sigma'\rangle \Downarrow \langle w;\sigma''\rangle}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, u);\sigma\rangle \Downarrow \langle w;\sigma''\rangle}$$

$$\frac{\begin{array}{c}\langle u;\sigma\rangle \Downarrow \langle \mathsf{wait}\, v_1\, v_2;\sigma'\rangle \\ \langle t[v_1/x, v_2/y, l/z];(\sigma', l\mapsto \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, \mathsf{adv}(v_2)))\rangle \Downarrow \langle v';\sigma''\rangle \qquad l = \mathsf{alloc}\,(\sigma')\end{array}}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, u);\sigma\rangle \Downarrow \langle v';\sigma''\rangle}$$

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

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

Fig. 4. Evaluation semantics.

$$\frac{\langle t;\eta\checkmark\rangle \Downarrow \langle v :: w; \eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle \mathsf{adv}\, w; \eta_L\rangle} \qquad \frac{\langle t;\eta\checkmark\rangle \Downarrow \langle \mathsf{wait}\, v\, w; \eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{adv}\, w; \eta_L\rangle} \qquad \frac{\langle t;\eta\checkmark\rangle \Downarrow \langle \mathsf{now}\, v; \eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_L\rangle}$$

$$\frac{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle t';\eta'\rangle}{\langle t;\eta;p\rangle \overset{\mathsf{in}_p\, v}{\Longrightarrow}_{\mathsf{F}} \langle t';\eta';p\rangle} \qquad \frac{\langle t;\eta\rangle \overset{\langle v,w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta'\rangle}{\langle t;\eta;1\rangle \overset{\mathsf{in}_2\, v}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{adv}\, w;\eta';2\rangle} \qquad \frac{\langle t;\eta\rangle \overset{\langle v,w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta'\rangle}{\langle t;\eta;2\rangle \overset{\mathsf{in}_1\, v}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{out}(\mathsf{adv}\, w);\eta';1\rangle}$$

Fig. 5. Step semantics for streams, until types and fair streams.

The operational semantics of the guarded fixed point combinator fix closely follows the intuition provided by its type: The fixed point $\mathsf{fix}\, x.t$ is unfolded by delaying it into the future and substituting it for x in t. However, since $\mathsf{fix}\, x.t$ is of type $\Box A$, it first has to be unboxed before the delay and boxed again afterwards.

The recursion principle for $\mathcal{U}$ types is similar to the primitive recursion principle one would obtain for an inductive type $\mu\alpha.B + (A \times \alpha)$. The difference to an $\mathcal{U}$ type, i.e., a type $\mu\alpha.B + (A \times \bigcirc\alpha)$, is that each recursive call $\mathsf{rec}_{\mathcal{U}}(\dots)$ is shifted one time step into the future by placing it in the heap. As opposed to fix, however, no additional unboxing and re-boxing is required.

4.2 Step Semantics

The *step semantics* given in Figure 5 describes the computation performed by a Lively RaTT program *over time*. The notation $\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle t';\eta'\rangle$ indicates the passage of one time step during which a configuration consisting of a stream program t and a heap η transitions to the program t' and heap η' emitting the output v. We give three separate step semantics for stream, until, and fair stream types, denoted $\Longrightarrow_{\mathsf{Str}}$, $\Longrightarrow_{\mathcal{U}}$, and $\Longrightarrow_{\mathsf{F}}$, respectively. In addition, we state our metatheoretic results: $\Longrightarrow_{\mathsf{Str}}$ is productive, $\Longrightarrow_{\mathcal{U}}$ is guaranteed to terminate, and $\Longrightarrow_{\mathsf{F}}$ indeed produces a *fair* stream. But we defer the proofs of these results to section 5.

A closed term t of type $\Box\mathsf{Str}(A)$ is meant to produce an infinite stream $v_1, v_2, v_3, \dots$ of values of type A in a step-by-step fashion:

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

Each step $\langle t_i;\eta_i\rangle \overset{v_i}{\Longrightarrow}_{\mathsf{Str}} \langle t_{i+1};\eta_{i+1}\rangle$ proceeds by first evaluating $\langle t_i;\eta_i\checkmark\rangle$ to $\left\langle v_i :: w; \eta_i'\checkmark\eta_{i+1}\right\rangle$, i.e., the head $v_i : A$ and the tail $w : \triangleright\mathsf{Str}(A)$ of the stream. Computation may then proceed in the next time step with the term $t_{i+1} = \mathsf{adv}\, w$ and heap η_{i+1} The old heap η_i', which consists of η_i possibly extended during evaluation of t_i, is garbage collected.

We can show that a term of type $\Box\mathsf{Str}(A)$ indeed produces such an infinite sequence of outputs. To state the productivity property of streams concisely, we restrict ourselves to streams over *value types*, which are described by the following grammar:

$$U, V ::= 1 \mid \mathsf{Nat} \mid U \times V \mid U + V$$

Theorem 4.1 (Productivity). If $\vdash t : \Box\mathsf{Str}(A)$, then there is an infinite sequence of reduction steps

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

Moreover, if A is a value type, then $\vdash v_i : A$ for all $i \geq 1$.

In particular, this means that the machine will never get stuck trying to access a heap location that has been garbage collected. In other words, the aggressive garbage collection of the heap used in the step semantics is safe.

Intuitively speaking, value types describe static, time independent data, and therefore exclude functions and modal types. Since A is a value type in the above theorem, we can give a concise characterisation of the output produced by the stream in terms of the syntactic typing $\vdash v_i : A$.

The operational semantics for until types proceeds similarly to stream types, but has an additional case for when the computation eventually halts by evaluating to a value of the form $\mathsf{now}\, v$.

We can show that a term of type $\Box(A \mathcal{U} B)$ produces a sequence of values of type A, but eventually halts with a value of type B:

Theorem 4.2 (Termination). If $\vdash t : \Box(A \mathcal{U} B)$, then there is a finite sequence of reduction steps

$$\langle \mathsf{unbox}\, t; \emptyset \rangle \overset{v_1}{\Longrightarrow}_{\mathcal{U}} \langle t_1; \eta_1 \rangle \overset{v_2}{\Longrightarrow}_{\mathcal{U}} \langle t_2; \eta_2 \rangle \overset{v_2}{\Longrightarrow}_{\mathcal{U}} \dots \overset{v_n}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_n \rangle$$

Moreover, if A and B are value types, then $\vdash v_i : A$ for all $0 < i < n$, and $\vdash v_n : B$.

The computation performed by a fair stream of type $\Box\mathsf{Fair}(\mathsf{A},\mathsf{B})$ requires a bit of additional bookkeeping: The state of the machine is represented by a triple $\langle t; \eta; p \rangle$ consisting of a term t, a heap η and a value $p \in \{1, 2\}$ that indicates the mode that the computation represented by t is in. If $p = 1$, then the most recent output was of type A, whereas $p = 2$ indicates that the most recent output was of type B. A fair execution thus means that the computation may not remain in the same mode indefinitely.

Theorem 4.3 (Liveness). If $\vdash t : \Box\mathsf{Fair}(\mathsf{A},\mathsf{B})$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{out}\,(\mathsf{unbox}\, t); \emptyset; 1 \rangle \overset{\mathsf{in}_{p_1} v_1}{\Longrightarrow}_{\mathsf{F}} \langle t_1; \eta_1; p_1 \rangle \overset{\mathsf{in}_{p_2} v_2}{\Longrightarrow}_{\mathsf{F}} \langle t_2; \eta_2; p_2 \rangle \overset{\mathsf{in}_{p_3} v_3}{\Longrightarrow}_{\mathsf{F}} \dots$$

such that for each $p \in \{1, 2\}$, we have that $p_i = p$ for infinitely many $i \geq 1$. Moreover, if A and B are value types, then $\vdash \mathsf{in}_{p_i} v_i : A + B$ for all $i \geq 1$.

As a special case of the fair stream type we obtain the type $\mathsf{Live}(A) = \mathsf{Fair}(1, \mathsf{A})$. Terms of this type do not produce a result at every time step but they will produce infinitely many results.

4.3 Reactive Step Semantics

The step semantics in section 4.2 captures closed computations without input from an external environment. To capture reactive computations, we give for each step semantics $\Longrightarrow_M$, a corresponding *reactive* step semantics in Figure 6. For instance, the reactive step semantics for streams may at each step consume some input v and produce an output v', which we denote by $\overset{v/v'}{\Longrightarrow}_{\mathsf{Str}}$.

To supply input to the computation, the machine configurations for the reactive step semantics are extended by an additional component l, which is the heap location from where the next input value can be retrieved. For instance, the reactive step semantics for streams takes a step $\langle t; \eta; l \rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle t'; \eta'; l' \rangle$ by placing the value $v :: l'$ in the heap location location l, where l' is a freshly allocated heap location that serves as a stand-in for the subsequent input. Assigning l' the dummy value $\langle\rangle$ will prevent the machine from allocating l' during evaluation of t. The reactive step semantics for $\mathcal{U}$-types and fair streams follow the same pattern.

Given a term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, and a sequence $v_1, v_2, \dots$ of values of type A, there is an infinite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0 \rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathsf{Str}} \langle t_1; \eta_1; l_1 \rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathsf{Str}} \langle t_2; \eta_2; l_2 \rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

$$\frac{\langle t; \eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle v' :: w; \eta_N \checkmark \eta_L, l' \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle \mathsf{adv}\, w; \eta_L; l'\rangle}$$

$$\frac{\langle t; \eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle \mathsf{wait}\, v'\, w; \eta_N \checkmark \eta_L, l \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{adv}\, w; \eta_L; l'\rangle}$$

$$\frac{\langle t; \eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle \mathsf{now}\, v'; \eta_N \checkmark \eta_L, l' \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_L; l'\rangle}$$

$$\frac{\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle t'; \eta'; l'\rangle}{\langle t; \eta; l; p\rangle \overset{v/\mathsf{in}_p\, v'}{\Longrightarrow}_{\mathsf{F}} \langle t'; \eta'; l'; p\rangle} \qquad \frac{\langle t; \eta; l\rangle \overset{v/\langle v', w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta'; l'\rangle}{\langle t; \eta; l; 1\rangle \overset{v/\mathsf{in}_2\, v'}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{adv}\, w; \eta'; l'; 2\rangle}$$

$$\frac{\langle t; \eta; l\rangle \overset{v/\langle v', w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta'; l'\rangle}{\langle t; \eta; l; 2\rangle \overset{v/\mathsf{in}_1\, v'}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{out}(\mathsf{adv}\, w); \eta'; l'; 1\rangle}$$

Fig. 6. Reactive step semantics for streams, until types and fair streams.

such that $\vdash v_i : B$ for all $i \geq 1$. The first term of the computation sets up the initial promise of an input by giving the term $\mathsf{unbox}\, t$ of type $\mathsf{Str}(A) \to \mathsf{Str}(B)$ the argument $\mathsf{adv}\, l_0$. The location l_0 is simply the first fresh heap location, i.e. $l_0 = \mathsf{alloc}\,(\emptyset)$; While $\mathsf{adv}\, l_0$ is not a well-typed term, it has the type $\mathsf{Str}(A)$ *semantically*, in the sense that $\mathsf{adv}\, l_0$ is a term in the logical relation $[\![\mathsf{Str}(A)]\!]$ that we construct in section 5.

Theorem 4.4 (Causality). Let $v_1, v_2, \ldots$ be an infinite sequence of values with $\vdash v_i : A$ for all $i \geq 1$.

(i) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathsf{Str}} \langle t_1; \eta_1; l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathsf{Str}} \langle t_2; \eta_2; l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

Moreover, if B is a value type, then $\vdash v_i' : B$ for all $i \geq 1$.

(ii) If $\vdash t : \Box(\mathsf{Str}(A) \to B\, \mathcal{U}\, C)$, then there is a finite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathcal{U}} \langle t_1; \eta_1; l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathcal{U}} \langle t_2; \eta_2; l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathcal{U}} \ldots \overset{v_n/v_n'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_n; l_n\rangle$$

Moreover, if B and C are value types, then $\vdash v_i' : B$ for all $0 < i < n$, and $\vdash v_n' : C$.

(iii) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Fair(B, C)})$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{out}\,(\mathsf{unbox}\, t\, (\mathsf{adv}\, l_0)); \emptyset; l_0; 1\rangle \overset{v_1/\mathsf{in}_{p_1} v_1'}{\Longrightarrow}_{\mathsf{F}} \langle t_1; \eta_1; l_1; p_1\rangle \overset{v_2/\mathsf{in}_{p_2} v_2'}{\Longrightarrow}_{\mathsf{F}} \langle t_2; \eta_2; l_2; p_2\rangle \overset{v_3/\mathsf{in}_{p_3} v_3'}{\Longrightarrow}_{\mathsf{F}} \ldots$$

such that for each $p \in \{1, 2\}$, we have that $p_i = p$ for infinitely many $i \geq 1$. Moreover, if B and C are value types, then $\vdash \mathsf{in}_{p_i} v_i' : B + C$ for all $i \geq 1$.

Since the operational semantics is deterministic, in each step $\langle t_i; \eta_i; l_i\rangle \overset{v_{i+1}/v_{i+1}'}{\Longrightarrow}_{\mathsf{Str}} \langle t_{i+1}; \eta_{i+1}; l_{i+1}\rangle$ the resulting output v_{i+1}' and new state of the computation $\langle t_{i+1}; \eta_{i+1}; l_{i+1}\rangle$ are uniquely determined by the previous state $\langle t_i; \eta_i; l_i\rangle$ and the input v_{i+1}. Thus, v_{i+1}' and $\langle t_{i+1}; \eta_{i+1}; l_{i+1}\rangle$ are independent of future inputs v_j with $j > i + 1$. The same is true for the corresponding reactive step semantics of $\mathcal{U}$-types and fair streams.

5 METATHEORY

In this section we show the soundness of the type system, which typically means that a well-typed term will never get stuck. However, we show a stronger, *semantic* type soundness property that will allow us to prove the operational properties detailed in section 4. 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 sets of "worlds" and defined inductively on the structure of the type A and world w. The proof of soundness is then reduced to a proof that $\vdash t : A$ implies $t \in [\![A]\!](w)$.

5.1 Worlds

To a first approximation, the worlds in our logical relation contain two ordinals $\alpha \leqslant \omega$ and $\beta < \omega \cdot 2$ and a store σ. The two ordinals are used to define the logical relation for recursive types, the first for temporal inductive types and the latter for step-indexed guarded recursive types. For guarded recursive types, we achieve this by defining $[\![\triangleright A]\!](\sigma, \alpha, \beta)$ in terms of $[\![A]\!](\sigma, \alpha, \beta')$ for strictly smaller β'. Since unfolding $\mathsf{Fix}\,\alpha.A$ introduces a $\triangleright$, the step index decreases for the recursive call. For the inductive types, we define $[\![A \,\mathcal{U}\, B]\!](\sigma, \alpha, \beta)$ in terms of $[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \alpha', \beta)$ where $\alpha' < \alpha$. Intuitively, α gives an upper limit to the number of unfoldings of the inductive type used in terms.

While this setup is sufficient for proving productivity, safety of garbage collection, termination, and liveness properties, it is not enough to capture causality. To characterise causality, the logical relation also needs to account for the possible inputs a given term may receive. We do this by further indexing the logical relation by a sequence of future inputs. To this end, we assume an infinite sequence of heaps $\eta_1; \eta_2; \ldots$, denoted $\overline{\eta}$, that describes the input that is received at each point in the future. The namespaces of all heaps in $\overline{\eta}$ and σ are assumed pairwise disjoint. The worlds in our logical relation are thus of the from $(\sigma, \overline{\eta}, \alpha, \beta)$.

As is standard for Kripke logical relations, our relation will be closed under moving to a bigger world. Worlds are ordered as follows: $(\sigma, \overline{\eta}, \alpha, \beta) \leqslant (\sigma', \overline{\eta}', \alpha', \beta')$ if $\sigma \sqsubseteq_{\checkmark} \sigma'$, $\overline{\eta} \sqsubseteq \overline{\eta}'$, $\alpha = \alpha'$ and $\beta' \leqslant \beta$. Here $\overline{\eta} \sqsubseteq \overline{\eta}'$ refers to the pointwise ordering on partial maps (assuming identity of namespaces) and $\sigma \sqsubseteq_{\checkmark} \sigma'$ is the extension of this with the rule $\eta_L \sqsubseteq_{\checkmark} \eta_N \checkmark \eta_L$ to a preorder. Note that the null store $\bullet$ is only related to itself under this ordering.

5.2 Support and Renamings

To prove closure of the Kripke semantics under store extensions, a similar property must be proved for the machine, i.e, if $\langle t; \sigma \rangle$ evaluates to a value and if $\sigma \sqsubseteq_{\checkmark} \sigma'$ then $\langle t; \sigma' \rangle$ evaluates to the same value. However, this statement is not entirely true, because the machine, when run in an extended state, may allocate different locations on the heap and the resulting values may differ correspondingly. To prove that this is the only way that the values can differ, we introduce notions of a renaming and support. Similar notions were used in the model by Krishnaswami [2013].

A renaming is a map $\phi : \mathsf{Loc} \to \mathsf{Loc}$ respecting name spaces. Such a map acts on terms by substitution, and this extends to heaps and stores by $\phi(\eta, l \mapsto t) = \phi(\eta), \phi(l) \mapsto \phi(t)$. We write $\phi : (t, \sigma, \overline{\eta}) \to (t', \sigma', \overline{\eta}')$ if $\phi(t) = t', \phi(\sigma) \sqsubseteq_{\checkmark} \sigma', \phi(\overline{\eta}) \sqsubseteq \overline{\eta}'$. Given a term t and a pair of a store and a heap sequence $(\sigma, \overline{\eta})$, we say that t is *supported* by $(\sigma, \overline{\eta})$, written $t \bowtie (\sigma, \overline{\eta})$, if whenever a location in t occurs in the namespaces of σ or $\overline{\eta}$, it must be in the domain of σ or $\overline{\eta}$, respectively. Given $(\sigma, \overline{\eta})$, we write $(\sigma, \overline{\eta})$ supported if all values in the codomains of σ and $\overline{\eta}$ are supported by $(\sigma, \overline{\eta})$. We restrict attention to Kripke worlds where $(\sigma, \overline{\eta})$ is supported.

5.3 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 that evaluate to elements in $\mathcal{V}[\![A]\!](w)$. The two relations are defined by mutual induction in Figure 7. More precisely, the two relations are defined by well-founded recursion by the lexicographic ordering on the tuple $(\beta, |A|, \alpha, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$\begin{gathered}|\alpha| = |\triangleright A| = |1| = |\mathsf{Nat}| = 1\\ |A \times B| = |A + B| = |A \,\mathcal{U}\, B| = |A \to B| = 1 + |A| + |B|\\ |\square A| = |\bigcirc A| = |\mathsf{Fix}\,\alpha.A| = 1 + |A|\end{gathered}$$

Note that since $|\triangleright(\mathsf{Fix}\ \alpha.A)| = |\alpha|$, the size of $A[\triangleright\mathsf{Fix}\,\alpha.A/\alpha]$ is strictly smaller than that of $\mathsf{Fix}\,\alpha.A$, which justifies the well-foundedness of recursive types. Note also that $\mathcal{V}[\![A \,\mathcal{U}\, B]\!](\sigma, \overline{\eta}, \alpha, \beta)$ is defined in terms of $\mathcal{V}[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$ for $\alpha' < \alpha$. To obtain well-foundedness, we would need $|\bigcirc(A \,\mathcal{U}\, B)| \leqslant |A \,\mathcal{U}\, B|$, which is not true. But this problem can be avoided by "inlining" the definition of $\mathcal{V}[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$, which is defined in terms of $\mathcal{T}[\![A \,\mathcal{U}\, B]\!](\sigma', \overline{\eta}', \alpha', \beta)$ where σ' and $\overline{\eta}'$ are appropriately modified. For the sake of readability, we will keep the definitions as given.

In the definition for $A \to B$ we explicitly add the closure properties for support, renaming and worlds. Further, we restrict the lambda abstractions to garbage collected stores. This reflects the typing rule for lambda abstractions, which requires Γ to be tick-free.

The definition of $\square A$ captures the notion of stability. All terms must be free of locations and able to evaluate safely with any input sequence and hence, in any future.

The value relations for $\triangleright A$ and $\bigcirc A$ differ only in the case where β is a limit ordinal. In the successor case, they encapsulate the soundness of garbage collection: The set $\mathcal{V}[\![m\ a]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta + 1), m \in \{\bigcirc, \triangleright\}$ contains all heap locations that can be read and executed safely in the next timestep. In particular, such terms must evaluate in the garbage collected store extended with the next set of external inputs. If β is a limit ordinal, $\bigcirc A$ has the same interpretation except that the index β is fixed. This is to ensure that inductive types have the correct global behaviour. On the other hand, $\triangleright A$ is defined to be the intersection of the interpretation at all smaller (β)-indices. This definition is needed for the interpretation of fixed points.

In the definition of $A \,\mathcal{U}\, B$ we see the use of the α-index to give an upper bound on the number of unfoldings used in the elements of the logical relation. In particular, if $\alpha = 0$, the relation contains only values of the form $\mathsf{now}\,v$ whereas if $\alpha > 0$, the relation also contain values of the form $\mathsf{wait}\,u\,w$ defined in terms of values from $\mathcal{V}[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$ where $\alpha' < \alpha$.

Our value and term interpretation is closed w.r.t the Kripke structure on $\sigma, \overline{\eta}$ and β and the value relation is upwards closed w.r.t α for $\mathcal{U}$-types.

Lemma 5.1 (Kripke Properties). *Given A, B and worlds $(\sigma, \overline{\eta}, \alpha, \beta), (\sigma', \overline{\eta}', \alpha', \beta')$ s.t $\sigma \sqsubseteq_{\checkmark} \sigma', \overline{\eta} \sqsubseteq \overline{\eta}', \alpha \leqslant \alpha'$ and $\beta' \leqslant \beta$ we have*

(1) $\mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![A]\!](\sigma', \overline{\eta}', \alpha, \beta')$
(2) $\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{T}[\![A]\!](\sigma', \overline{\eta}', \alpha, \beta')$
(3) $\mathcal{V}[\![A \,\mathcal{U}\, B]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![A \,\mathcal{U}\, B]\!](\sigma, \overline{\eta}, \alpha', \beta)$

As stated above we treat $\bigcirc$ as a sub-modality of $\triangleright$ and this is expressed semantically in the following lemma:

$$
\begin{aligned}
\mathcal{V}[\![1]\!](w) &= \{\langle\rangle\}, \\
\mathcal{V}[\![\mathsf{Nat}]\!](w) &= \{\mathsf{suc}^n\, 0 \mid n \in \mathbb{N}\}, \\
\mathcal{V}[\![A \times B]\!](w) &= \left\{(v_1, v_2) \,\middle|\, v_1 \in \mathcal{V}[\![A]\!](w) \wedge v_2 \in \mathcal{V}[\![B]\!](w)\right\}, \\
\mathcal{V}[\![A + B]\!](w) &= \left\{\mathsf{in}_1\, v \,\middle|\, v \in \mathcal{V}[\![A]\!](w)\right\} \cup \left\{\mathsf{in}_2\, v \,\middle|\, v \in \mathcal{V}[\![B]\!](w)\right\} \\
\mathcal{V}[\![A \to B]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \big\{\lambda x.t \,\big|\, t \bowtie (\sigma, \overline{\eta}) \wedge \forall \beta' \leq \beta. \forall \psi : (t, \sigma, \overline{\eta}) \to (t', \sigma', \overline{\eta}'). \\
&\qquad \forall v \in \mathcal{V}[\![A]\!](\mathsf{gc}\,(\sigma'), \overline{\eta}', \alpha, \beta').t'[v/x] \in \mathcal{T}[\![B]\!](\mathsf{gc}\,(\sigma'), \overline{\eta}', \alpha, \beta')\big\} \\
\mathcal{V}[\![\square A]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{t \,\middle|\, \forall \overline{\eta}'.\mathsf{unbox}\, t \in \mathcal{T}[\![A]\!](\emptyset, \overline{\eta}', \alpha, \beta) \wedge t \text{ location-free}\right\} \\
\mathcal{V}[\![\bigcirc A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) &= \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) & \beta = 0 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta')\right\} & \beta = \beta' + 1 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta)\right\} & \beta \text{ limit ordinal} \wedge \sigma \neq \bullet \end{cases} \\
\mathcal{V}[\![\rhd A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) &= \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) & \beta = 0 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta')\right\} & \beta = \beta' + 1 \wedge \sigma \neq \bullet \\ \bigcap_{\beta' < \beta} \mathcal{V}[\![\rhd A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta') & \beta \text{ limit ordinal} \wedge \sigma \neq \bullet \end{cases} \\
\mathcal{V}[\![\mathsf{Fix}\, \alpha.A]\!](w) &= \left\{\mathsf{into}(v) \,\middle|\, v \in \mathcal{V}[\![A[\rhd(\mathsf{Fix}\, \alpha.A)/\alpha]]\!](w)\right\} \\
\mathcal{V}[\![A \,\mathcal{U}\, B]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{\mathsf{now}\, v \,\middle|\, v \in \mathcal{V}[\![B]\!](\sigma, \overline{\eta}, \omega, \beta)\right\} \cup \\
&\quad \big\{\mathsf{wait}\, v\, w \,\big|\, v \in \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta) \\
&\qquad \wedge \exists \alpha' < \alpha. w \in \mathcal{V}[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \overline{\eta}, \alpha', \beta)\big\} \\
\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{t \,\middle|\, t \bowtie (\sigma, \overline{\eta}) \wedge \exists \sigma', v.\, \langle t; \sigma\rangle \Downarrow \langle v; \sigma'\rangle \wedge v \in \mathcal{V}[\![A]\!](\sigma', \overline{\eta}, \alpha, \beta)\right\}
\end{aligned}
$$

Garbage collection: $\mathsf{gc}\,(\bullet) = \bullet \qquad \mathsf{gc}\,(\eta_L) = \eta_L \qquad \mathsf{gc}\,(\eta_N \checkmark \eta_L) = \eta_L$

Fig. 7. Logical Relation.

Lemma 5.2 (Sub-modality). *Given A and world $(\sigma, \overline{\eta}, \alpha, \beta)$ then*

$$\mathcal{V}[\![\bigcirc A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![\rhd A]\!](\sigma, \overline{\eta}, \alpha, \beta)$$

Proof. Follows by transfinite induction on β. □

The next lemma justifies the terminology 'limit types', by showing that the interpretation of these at limit ordinals is the intersection of the interpretations at the ordinals below. In category theoretic terms, the intersection is a limit, and such a type is a sheaf [MacLane and Moerdijk 2012].

Lemma 5.3 (Limit Types). *If A limit and β is a limit ordinal, then*

$$\bigcap_{\beta' < \beta} \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta') = \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \qquad \bigcap_{\beta' < \beta} \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta') = \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta)$$

Proof. In the first equality, the inclusion from right to left follows from Lemma 5.1, and the other inclusion is proved by induction on A. The second equality then follows from the first. □

In the special case where A is a limit type, $\bigcirc$ and $\rhd$ do in fact coincide:

Corollary 5.4 (Sub-modality at limit). *Given A and a world w s.t. A limit, then*

$$\mathcal{V}[\![\bigcirc A]\!](w) = \mathcal{V}[\![\rhd A]\!](w)$$

$$
\begin{aligned}
\mathcal{C}[\![\cdot]\!](\bullet, \overline{\eta}, \beta) &= \{*\} \\
\mathcal{C}[\![\Gamma, x : A]\!](\sigma, \overline{\eta}, \beta) &= \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in \mathcal{C}[\![\Gamma]\!](\sigma, \overline{\eta}, \beta), v \in \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta) \right\} \\
\mathcal{C}[\![\Gamma, \checkmark_{\triangleright}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) &= \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1) \\
\mathcal{C}[\![\Gamma, \checkmark_{\bigcirc}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) &= \begin{cases} \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta) & \beta \text{ limit ordinal} \\ \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1) & \text{otherwise} \end{cases} \\
\mathcal{C}[\![\Gamma, \sharp]\!](\sigma, \overline{\eta}, \beta) &= \bigcup_{\overline{\eta}'} \mathcal{C}[\![\Gamma]\!](\bullet, \overline{\eta}', \beta) \qquad\qquad \sigma \neq \bullet
\end{aligned}
$$

Fig. 8. Context Relation

Proof. One inclusion always holds by Lemma 5.2, and the two sets are equal by definition except when β is a limit ordinal. In that case, by Lemma 5.3 it suffices to show that if $v \in \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta)$ then $\mathsf{adv}\ v \in \mathcal{T}[\![A]\!]((\mathsf{gc}\,(\sigma) \checkmark \eta); \overline{\eta}, \alpha, \beta')$ for all $\beta' < \beta$, which follows from $v \in \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta' + 1)$ □

Finally, we obtain the semantic soundness of the language phrased as the following fundamental property of the logical relation $\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$.

Theorem 5.5 (Fundamental Property). If $\Gamma \vdash t : A$ and $\gamma \in \mathcal{C}[\![\Gamma]\!](\sigma, \overline{\eta}, \beta)$, then $t\gamma \in \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$.

Here $\mathcal{C}[\![\Gamma]\!](\sigma, \overline{\eta}, \beta)$ refers to the logical relation for typing contexts defined in Figure 8. Note the cases for $\Gamma, \checkmark_m$, which captures the intuition that variables occurring before $\checkmark_m$ arrive one time step before those to the right. Again $\bigcirc$ and $\triangleright$ differ only when β is a limit ordinal. Cases not mentioned in the figure (such as $\mathcal{C}[\![\cdot]\!](\sigma, \overline{\eta}, \beta)$ for $\sigma \neq \bullet$) are interpreted as the empty set. The theorem is proved by a lengthy but entirely standard induction on the typing relation $\Gamma \vdash t : A$.

As an easy consequence of the fundamental property and the fact the empty substitution is an element of $\mathcal{C}[\![\cdot, \sharp]\!](\sigma, \overline{\eta}, \beta)$ for any store σ and input sequence $\overline{\eta}$, we have the following property that we shall use to prove Lively RaTT's operational properties:

Corollary 5.6 (Fundamental Property). *If $\sharp \vdash t : A$, then $t \in \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$ for all $\sigma, \overline{\eta}$ and β.*

5.4 Productivity, Termination, Liveness & Causality

In this section we demonstrate how we apply the fundamental property of the logical relation to prove the operational properties of Lively RaTT that we presented in section 4.2 and section 4.3. We have formulated these operational properties in terms of value types, so that we can use the following correspondence between semantic and syntactic typing:

Lemma 5.7. *Given any world w, value type A, and value v, we have that $v \in \mathcal{V}[\![A]\!](w)$ iff $\vdash v : A$.*

5.4.1 Productivity. We start with the productivity property of streams of type $\mathsf{Str}(A)$.

Given a type A and ordinal β, we define the following set of machine configurations for $\Longrightarrow_{\mathsf{Str}}$ that are safe according to the logical relation:

$$
S(A, \beta) = \left\{ \langle t; \eta \rangle \,\middle|\, t \in \mathcal{T}[\![\mathsf{Str}(A)]\!](\eta \checkmark, \overline{\emptyset}, \omega, \beta) \right\}
$$

where we use the notation $\overline{\emptyset}$ for the sequence of empty heaps with the appropriate namespace.

Intuitively speaking, a machine configuration c in $S(A, \beta)$ will be safe to execute for the next β steps of $\Longrightarrow_{\mathsf{Str}}$ and produces output of type A. We formulate the essence of the productivity property of such a stream as follows:

Lemma 5.8 (Productivity). *If $\langle t; \eta \rangle \in S(A, \beta + 1)$, then there are $\langle t'; \eta' \rangle \in S(A, \beta)$ and $v \in \mathcal{V}[\![A]\!](\eta', \overline{\emptyset}, \omega, \beta + 1)$ such that $\langle t; \eta \rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle t'; \eta' \rangle$.*

In each step of a stream computation, we count down by one on the step index β and produce an output v of semantic type A:

Proof of Theorem 4.1. By Corollary 5.6 we have that $\langle \mathsf{unbox}\, t; \emptyset \rangle \in S(A, \beta)$ for any β. Using Lemma 5.8, we can thus extend any finite reduction sequence

$$\langle \mathsf{unbox}\, t; \emptyset \rangle \overset{v_1}{\Longrightarrow}_{\mathsf{Str}} c_1 \overset{v_2}{\Longrightarrow}_{\mathsf{Str}} c_2 \overset{v_3}{\Longrightarrow}_{\mathsf{Str}} \cdots \overset{v_n}{\Longrightarrow}_{\mathsf{Str}} c_n$$

by an additional reduction step $c_n \overset{v_{n+1}}{\Longrightarrow}_{\mathsf{Str}} c_{n+1}$. Since $\Longrightarrow_{\mathsf{Str}}$ is deterministic, this uniquely defines the desired infinite reduction. Moreover, given that A is a value type, $\vdash v_i : A$ follows for all $i \geq 1$ by Lemma 5.7. □

5.4.2 Termination. Analogously to the set of machine states $S(A, \beta)$ for stream types, we define the following set $U(A, B, \alpha, \beta)$ for until types:

$$U(A, B, \alpha, \beta) = \left\{ \langle t; \eta \rangle \,\middle|\, t \in \mathcal{T}[\![A \,\mathcal{U}\, B]\!](\eta\checkmark, \overline{\emptyset}, \alpha, \beta) \right\}$$

This definition allows us to state the essence of the termination property for until types as follows:

Lemma 5.9 (Termination). *Given $\langle t; \eta \rangle \in U(A, B, \alpha, \beta)$, one of the following two statements holds:*

(a) There are t', η', $\alpha' < \alpha$, and $v \in \mathcal{V}[\![A]\!](\eta', \overline{\emptyset}, \omega, \beta)$ such that

$$\langle t; \eta \rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle t'; \eta' \rangle \text{ and if } \beta > 0 \text{ then } \langle t'; \eta' \rangle \in U(A, B, \alpha', \beta - 1)$$

where $\beta - 1 = \beta'$ if $\beta = \beta' + 1$ and otherwise $\beta - 1 = \beta$.

(b) There is some $v \in \mathcal{V}[\![B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ such that $\langle t; \eta \rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta' \rangle$.

Theorem 4.2 is now an easy consequence of the above lemma and the fundamental property of the logical relation.

Proof of Theorem 4.2. By Corollary 5.6 $\mathsf{unbox}\, t \in U(A, B, \omega, \omega)$, and by Lemma 5.9 we can construct the desired sequence of reductions. Since the index α strictly decreases each time we take a step of the form (a), the sequence must eventually terminate with a step of the form (b). Moreover, by Lemma 5.7, the output values v_i have the desired type given that A and B are value types. □

5.4.3 Liveness. Recall that the step semantics of fair streams $\Longrightarrow_{\mathsf{F}}$ is a machine whose configurations are tuples $\langle t; \eta; p \rangle$, where $p \in \{1, 2\}$ indicates the current mode of the computation. The behaviour of the different modes is captured by the following definition of the set $F(A, B, \alpha, \beta)$ of such pairs:

$$\begin{aligned} F(A, B, \alpha, \beta) = \ & \{ \langle t; \eta; 1 \rangle \mid \langle t; \eta \rangle \in U(A, B \times {\rhd}(B \,\mathcal{U}\, (A \times {\rhd}\mathsf{Fair}(\mathsf{A}, \mathsf{B}))), \alpha, \beta) \} \\ & \cup \{ \langle t; \eta; 2 \rangle \mid \langle t; \eta \rangle \in U(B, A \times {\rhd}\mathsf{Fair}(\mathsf{A}, \mathsf{B}), \alpha, \beta) \} \end{aligned}$$

That is, if $p = 1$, then t belongs semantically to an until type $A \,\mathcal{U}\, (B \times {\rhd}\mathsf{Fair}'(\mathsf{B}, \mathsf{A}))$, and otherwise t belongs to $B \,\mathcal{U}\, (A \times {\rhd}\mathsf{Fair}(\mathsf{A}, \mathsf{B}))$.

With this characterisation, we can formulate the essence of the liveness property for fair streams:

Lemma 5.10 (Liveness). *Given $\langle t; \eta; p \rangle \in F(A, B, \alpha, \beta)$, one of the following statements is true:*

(a) there are $t', \eta', \alpha' < \alpha$, and $v \in \mathcal{V}[\![A+B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ such that

$$\langle t; \eta; p \rangle \overset{v}{\Longrightarrow}_{\mathsf{F}} \langle t'; \eta'; p \rangle \text{ and } \langle t'; \eta'; p \rangle \in F(A, B, \alpha', \beta') \text{ for all } \beta' < \beta.$$

(b) there are t', η' and $v \in \mathcal{V}[\![A+B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ such that

$$\langle t; \eta; p \rangle \overset{v}{\Longrightarrow}_{\mathsf{F}} \langle t'; \eta'; 3-p \rangle \text{ and } \langle t'; \eta'; 3-p \rangle \in F(A, B, \omega, \beta') \text{ for all } \beta' < \beta.$$

The liveness result is now an easy consequence of the above lemma and the fundamental property:

Proof of Theorem 4.3. By Theorem 5.6 $\mathsf{out}\,(\mathsf{unbox}\,t) \in F(A, B, \omega, \omega+n)$ for any n. Using Lemma 5.10, we can then show that we can extend any finite reduction sequence

$$\langle \mathsf{out}\,(\mathsf{unbox}\,t); \eta_0; 1 \rangle \overset{\mathsf{in}_{p_1} v_1}{\Longrightarrow}_{\mathsf{F}} \langle t_1; \eta_i; p_1 \rangle \overset{\mathsf{in}_{p_2} v_2}{\Longrightarrow}_{\mathsf{F}} \langle t_2; \eta_0; p_2 \rangle \overset{\mathsf{in}_{p_3} v_3}{\Longrightarrow}_{\mathsf{F}} \dots \overset{\mathsf{in}_{p_n} v_n}{\Longrightarrow}_{\mathsf{F}} \langle t_n; \eta_n; p_n \rangle$$

with $\langle t_n; \eta_n; p_n \rangle \overset{\mathsf{in}_{p_{n+1}} v_{n+1}}{\Longrightarrow}_{\mathsf{F}} \langle t_{n+1}; \eta_{n+1}; p_{n+1} \rangle$ so that $\langle t_{n+1}; \eta_{n+1}; p_{n+1} \rangle \in F(A, B, \omega, \omega)$. Since $\Longrightarrow_{\mathsf{F}}$ is deterministic, this defines the desired infinite sequence of reductions. Moreover, since $\langle t_i; \eta_i; p_i \rangle \in F(A, B, \omega, \omega)$ for each i, and the index α decreases for every step of the form (a), we know that only finitely many reduction steps after $\langle t_i; \eta_i; p_i \rangle$ are of the form (a). Thus, there is a $j \geq i$ with $p_j \neq p_i$. In addition, given that A and B are value types, so is $A + B$, and we thus obtain by Lemma 5.7 that $\vdash \mathsf{in}_{p_i} v_i : A + B$ for all $i \geq 1$. □

5.4.4 *Causality.* We conclude by sketching the proofs for the corresponding operational properties for the reactive step semantics. The proof idea is the same but instead of setting heaps in $\overline{\eta}$ to the empty heap, we construct $\overline{\eta}$ so that it contains the input stream.

Recall that after each step of the (reactive) step semantics, the machine starts a new empty 'later' heap. Each heap comes with its own namespace. Let l_i be the location that is picked as the first location by the allocator after i steps of the (reactive) step semantics, i.e. $\mathsf{alloc}\,(\eta_i) = l_i$ where η_i is the empty heap after i steps. Given a value v and number $i \geq 0$, we write η^i_v for the heap $l_i \mapsto v :: l_{i+1}$. We further define the following set of heap sequences:

$$H(A, i) = \left\{ \eta^i_{v_i}; \eta^{i+1}_{v_{i+1}}; \dots \,\middle|\, \forall j \geq i. \vdash v_j : A \right\}$$

The locations l_i are used to feed input to the reactive step semantics. The following lemma shows that each l_i has the right semantic type:

Lemma 5.11. *For all $i \geq 0$, $\vdash v : A$, and $\overline{\eta} \in H(A, i+1)$, we have that $l_i \in \mathcal{V}[\![\triangleright(\mathsf{Str}(A))]\!](\eta^i_v, \overline{\eta}, \omega, \beta)$.*

The lemma is proved by a straightforward induction on β using Corollary 5.6.

We can now prove variants of Lemma 5.8, 5.9, and 5.10 for the reactive step semantics. For the reactive step semantics of streams $\overset{/}{\Longrightarrow}_{\mathsf{Str}}$, we define the corresponding set of machine configurations that are safe according to the logical relation as follows:

$$S^R(\overline{\eta}, B, \beta) = \left\{ \langle t; \eta; l_i \rangle \,\middle|\, \overline{\eta} = \eta^i_v; \eta^{i+1}_w; \overline{\eta}' \wedge t \in \mathcal{T}[\![\mathsf{Str}(B)]\!]((\eta, \eta^i_v \checkmark \eta^{i+1}_w), \overline{\eta}', \omega, \beta) \right\}$$

This construction takes as additional parameter $\overline{\eta}$ drawn from $H(A, i)$ which represents future input. We can then formulate the corresponding productivity property as follows:

Lemma 5.12. *Given $\langle t; \eta; l_i \rangle \in S^R((\eta^i_v; \overline{\eta}), B, \beta+1)$ and $\overline{\eta} \in H(A, i+1)$, then there are $\langle t'; \eta'; l_{i+1} \rangle \in S^R(\overline{\eta}, B, \beta)$ and v' such that*

$$\langle t; \eta; l_i \rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle t'; \eta'; l_{i+1} \rangle .$$

Moreover, if B is a value type then $\vdash v' : B$.

The constructions for $\overset{/}{\Longrightarrow}_{\mathcal{U}}$ and $\overset{/}{\Longrightarrow}_{\mathsf{F}}$ are analogous and we can then prove the causality property of the reactive step semantics.

Proof of Theorem 4.4. We give the proof for part (i) of the theorem. Part (ii) and (iii) follow by a similar adaptation of the proofs of Theorems 4.2 and 4.3, respectively. By Corollary 5.6, $\mathsf{unbox}\, t \in \mathcal{T}[\![\mathsf{Str}(A) \to \mathsf{Str}(B)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$ for all β and for $\overline{\eta} = \eta^2_{v_3}; \eta^3_{v_4}; \ldots$. By Lemma 5.11 $\mathsf{adv}\, l_0 \in \mathcal{T}[\![\mathsf{Str}(A)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$ and thus $\mathsf{unbox}\, t\, (\mathsf{adv}\, l_0) \in \mathcal{T}[\![\mathsf{Str}(B)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$. Hence, we have $\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0 \rangle \in U(\eta^0_{v_1}; \eta^1_{v_2}; \overline{\eta}, B, \beta)$ for any β. Similarly to the proof of Theorem 4.1, we can then use Lemma 5.12 to construct the desired infinite sequence of reduction steps. □

6 RELATED WORK

The work by Cave et al. [2014] mentioned in the introduction defines a language with a modal operator $\bigcirc$ as well as inductive and coinductive types, but no guarded fixed points. They define a family of reduction relations indexed by ordinals up to and including ω. The relations corresponding to finite ordinals describe reductions up to finitely many steps, and the one at ω describes global behaviour. They give an interpretation of types as predicates on values indexed by ordinals up to and including ω, and similarly to our interpretation of types, the interpretation of $\bigcirc A$ at ω refers to the interpretation of A also at ω. Using this they prove strong normalisation, and sketch proofs of causality, productivity and liveness, but they do not prove lack of space leaks as done here. The motivation for omitting the guarded fixed point operator is exactly the observation mentioned in the introduction that these equate inductive and coinductive types. Instead, programming with coinductive types like streams must be done by coiteration. The present paper shows how to refine the modal type system to combine the type system of LTL with the power of the fixed point operator, gaining simplicity in programming and productivity checking. The language of Cave et al. [2014] has more general inductive and coinductive types than Lively RaTT (but not general guarded recursive types), see discussion in section 7. The idea of transfinite step indexing as used both here and by Cave et al. [2014], has also been used to model countable non-determinism [Bizjak et al. 2014] and distinguishing between logical and and concrete steps in program verification [Svendsen et al. 2016].

Jeffrey [2012] and Jeltsch [2012] independently discovered the connection between FRP and LTL. Jeltsch [2012, 2013] studied a category theoretic common notion of models of LTL and FRP. Jeffrey [2012] defined a language for FRP as an abstraction of a model defined in a functional programming language. Signals are defined directly as time-dependent values and LTL types are defined by quantifying over time. While the native function space of the language contains all signal functions, a type of causal functions is definable in the language. In later work, Jeffrey [2014] extends modal FRP with heterogeneous stream types, i.e., streams of elements whose types are given by a stream of types, and use this to encode past-time LTL. Unlike the present work, neither Jeltsch, nor Jeffrey define an operational semantics of programs, and therefore prove no operational metatheoretical results.

To our knowledge, the first work to define a modal type theory for FRP with a guarded fixed point operator is that of Krishnaswami and Benton [2011]. This line of work also studies type systems for eliminating implicit space and time leaks. Krishnaswami et al. [2012] use linear types to statically bound the size of the dataflow graph generated by a reactive program, while Krishnaswami [2013] defines a simpler type system, but rules out space leaks using the techniques also used in the present paper. Bahr et al. [2019] recast this work in the setting of Simply RaTT, which unlike Krishnaswami [2013] uses Fitch style for programming with modal types, and extend these results by identifying and eliminating a type of time leaks stemming from fixed points.

The guarded fixed point operator was first suggested by Nakano [2000] and has since received much attention in logics for program verification because it can be used as a synthetic approach [Appel et al. 2007; Birkedal et al. 2011] to step-indexing [Appel and McAllester 2001]. Moreover, combining this with a notion of quantification over clocks [Atkey and McBride 2013] or a constant modality [Clouston et al. 2015] one can use guarded recursion to encode coinduction. Guarded recursion forms part of the foundation of the framework Iris [Jung et al. 2015] for higher-order concurrent separation logic in Coq, and a number of dependent type theories with guarded recursion have been defined [Bahr et al. 2017; Birkedal et al. 2019; Bizjak et al. 2016]. In the simply typed setting Guatto [2018] extends this with a notion of time warps. The combination of guarded recursion and higher inductive types [Univalent Foundations Program 2013] has also been used for modelling process calculi [Møgelberg and Veltri 2019; Veltri and Vezzosi 2020]. Although related to the modal FRP calculi, these systems are usually much more expressive, since space and time leaks are ignored in their design. For example, they all include an operation $A \to \triangleright A$ transporting data into the future, a known source of space leaks.

7 CONCLUSION AND FUTURE WORK

This paper shows how guarded fixed points can be combined with liveness properties in modal FRP. While properties such as termination, liveness and fairness are perhaps beyond the scope of properties traditionally expressed in simply typed programming languages, they could naturally occur as parts of program specifications in dependently typed languages and proof assistants. We therefore view Lively RaTT as a conceptual stepping stone towards a dependently typed language for reactive programming.

The results of this paper have been presented in the setting of functional reactive programming, but we expect that the ideas will be relevant also in the setting of guarded recursion as described in section 6. In these settings, the fact that inductive and coinductive types coincide means that termination cannot be expressed directly. This leads to limitations in the setting of program verification, e.g., when defining notions such as weak bisimulation for programs [Møgelberg and Paviotti 2019] and processes. We expect that the tools developed here can be used in this respect once this work has been adapted to guarded recursion and extended to dependent types.

Future work also includes extending Lively RaTT with general classes of inductive types. Note that as seen here one must distinguish between ordinary inductive types such as Nat and temporal ones such as the $\mathcal{U}$ types, where the recursion involves time steps and the recursors therefore need to be stable. The temporal inductive types should be defined as a class of strictly positive inductive types where the recursion variable appears under a $\bigcirc$, generalising the $\mathcal{U}$ types used here. One could likewise add a class of coinductive types in the ordinary sense, but the temporal coinductive types are subsumed by the guarded recursive types.

ACKNOWLEDGMENTS

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

A ELABORATING SURFACE SYNTAX TO CORE CALCULUS

Throughout the paper we use syntactic sugar for writing Lively RaTT programs. In this appendix we give an overview how this surface syntax can be elaborated into the core calculus.

First, the use of pattern matching in function definitions is translated into (nested) case expressions in a standard way (including inlining shorthands such as $\mathsf{wait}_{\mathsf{Ev}}$). For example, the two clauses defining $bind_{\mathsf{Ev}}$ on page 8 are elaborated into the following single clause:

$$\begin{aligned} \mathit{bind}_{\mathsf{Ev}}\, f \sharp x = \text{case } x \text{ of } & \mathsf{into}\ (\mathsf{in}_1\ a)\,.\,(\mathsf{unbox}\ f)\ a \\ & \mathsf{into}\ (\mathsf{in}_2\ e)\,.\,\mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e)) \end{aligned}$$

The resulting general case expressions with nested pattern matching are then transformed step-by-step into the elimination forms of the core calculus. For example the above case expression is transformed into:

$$\begin{aligned} \text{case } \mathsf{out}\ x \text{ of } & \mathsf{in}_1\ a\,.\,(\mathsf{unbox}\ f)\ a \\ & \mathsf{in}_2\ e\,.\,\mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e)) \end{aligned}$$

Once these transformations are performed, all function definitions consist of a single clause (non-recursive functions and guarded recursive functions) or two clauses (functions defined by natural number or $\mathcal{U}$ recursion). We consider each of these cases in turn.

A guarded recursive function definition is of the form

$$f\, x_1 \dots x_n \sharp y_1 \dots y_m = t\,[f\, x_1 \dots x_n / r]$$

where f may not occur freely in t. This function definition is elaborated into the following term:

$$f = \lambda x_1. \dots \lambda x_n.\mathsf{fix}\, r.\lambda y_1. \dots \lambda y_m.t$$

Mutual guarded recursive function definitions define several functions simultaneously. We consider the case of two mutually guarded recursive functions, with the general case following in a similar manner:

$$\begin{aligned} f_1\, x_1 \dots x_n \sharp y_1 \dots y_m &= t_1\,[f_1\, x_1 \dots x_n / r_1, f_2\, x_1 \dots x_n / r_2] \\ f_2\, x_1 \dots x_n \sharp z_l \dots z_l &= t_2\,[f_1\, x_1 \dots x_n / r_1, f_2\, x_1 \dots x_n / r_2] \end{aligned}$$

where t_1 and t_2 do not contain free occurrences of f_1 or f_2. This definition is then transformed into

$$\begin{aligned} f &= \mathsf{fix}\, r.\,\left\langle \lambda y_1. \dots \lambda y_m.t_1 \left[\pi_1^{\Box}\, r/r_1, \pi_2^{\Box}\, r/r_2\right], \lambda z_1. \dots \lambda z_m.t_2 \left[\pi_1^{\Box}\, r/r_1, \pi_2^{\Box}\, r/r_2\right]\right\rangle \\ f_1 &= \lambda x_1. \dots \lambda x_n.\mathsf{box}(\pi_1(\mathsf{unbox}\, f)) \\ f_2 &= \lambda x_1. \dots \lambda x_n.\mathsf{box}(\pi_2(\mathsf{unbox}\, f)) \end{aligned}$$

where $\pi_i^{\Box} : \Box(\rhd(A_1 \times A_2)) \to \Box(\rhd A_i)$, for $i \in \{1, 2\}$, is defined by

$$\pi_i^{\Box} = \lambda x.\mathsf{box}(\mathsf{delay}(\pi_i(\mathsf{adv}(\mathsf{unbox}\, x))))$$

Functions defined by natural number recursion are of the form:

$$\begin{aligned} f\, 0 \quad\quad\;\; x_1 \dots x_n \sharp y_1 \dots y_m &= s \\ f\, (\mathsf{suc}\, z)\, x_1 \dots x_n \sharp y_1 \dots y_m &= t\,[f\, z/r] \end{aligned}$$

where f may not occur freely in t. This definition is elaborated into the term

$$f = \lambda x.\mathsf{rec}_{\mathsf{Nat}}(\lambda x_1. \dots \lambda x_n.\mathsf{box}\,(\lambda y_1. \dots \lambda y_m.s), z\, r.\lambda x_1. \dots \lambda x_n.\mathsf{box}\,(\lambda y_1. \dots \lambda y_m.t), x)$$

In a recursive definition without $\sharp$, the transformation is the same but omits box.

Functions defined by $\mathcal{U}$ recursion are of the form:

$$\begin{aligned} f\, (\mathsf{now}\, x) \quad x_1 \dots x_n &= s \\ f\, (\mathsf{wait}\, x\, t)\, x_1 \dots x_n &= t\,[f\, y/r] \end{aligned}$$

where f may not occur freely in t. This definition is elaborated into the term

$$f = \lambda z.\mathsf{rec}_{\mathcal{U}}(x.\lambda x_1. \dots \lambda x_n.s, x\, y\, r.\lambda x_1. \dots \lambda x_n.t, z)$$

REFERENCES

Andreas Abel and Brigitte Pientka. 2013. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In *Proceedings ICFP 2013*. ACM, New York, NY, USA, 185–196. https://doi.org/10.1145/2500365.2500591

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *PACMPL* 1, ICFP (2017), 33:1–33:30. https://doi.org/10.1145/3110277

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.

Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/1190215.1190235

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. *ACM SIGPLAN Notices* 48, 9 (2013), 197–208. https://doi.org/10.1145/2500365.2500597

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

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–27. https://doi.org/10.1145/3341713

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2019. Guarded Cubical Type Theory. *Journal of Automated Reasoning* 63, 2 (01 Aug 2019), 211–253. https://doi.org/10.1007/s10817-018-9471-7

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

Aleš Bizjak, Lars Birkedal, and Marino Miculan. 2014. A Model of Countable Nondeterminism in Guarded Type Theory. In *Rewriting and Typed Lambda Calculi*, Gilles Dowek (Ed.). Springer International Publishing, Cham, 108–123. https://doi.org/10.1007/978-3-319-08918-8_8

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. https://doi.org/10.1007/978-3-662-49630-5_2

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.). Springer International Publishing, Cham, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421. https://doi.org/10.1007/978-3-662-46678-0_26

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

Frederic Benton Fitch. 1952. *Symbolic logic, an introduction*. Ronald Press Co., New York, NY, USA. https://doi.org/10.2307/2266614

Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, New York, NY, USA, 482–491. https://doi.org/10.1145/3209108.3209148

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

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. 2012. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science* 286 (2012), 229–242. https://doi.org/10.1016/j.entcs.2012.08.015

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650. https://doi.org/10.1145/2775051.2676980

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

Saunders MacLane and Ieke Moerdijk. 2012. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, New York, NY, USA. https://doi.org/10.1007/978-1-4612-0927-0

Rasmus E Møgelberg and Marco Paviotti. 2019. Denotational semantics of recursive types in synthetic guarded domain theory. *Mathematical Structures in Computer Science* 29, 3 (2019), 465–510. https://doi.org/10.1017/S0960129518000087

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as path type for guarded recursive types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290317

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 (SFCS '77)*. IEEE Computer Society, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE, New Orleans, LA, USA, 233–242. https://doi.org/10.1109/LICS.2013.29

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/3372885.3373814