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.

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. 2020. Diamonds are not forever: Liveness in reactive programming with guarded recursion. 1, 1 (March 2020), 26 pages. https://doi.org/10.1145/nmmmmnn.nnmm

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 is unfortunate, since much of the most critical software currently in use is reactive. The goal of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/3-ART $15.00
https://doi.org/10.1145/nmmmmnn.nnmm

, Vol. 1, No. 1, Article . Publication date: March 2020.
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 \( \text{Str}(A) \equiv A \times \text{Str}(A) \), 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 \( \forall A \) 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 \( \text{Str}(A) \equiv A \times \forall \text{Str}(A) \) 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 \( \text{Ev}(A) \equiv A + \forall \text{Ev}(A) \), stating that an event can either occur now, or at some point in the future. Types such as \( \text{Str}(A) \) and \( \text{Ev}(A) \) satisfying type equations in which the recursion variable is guarded by a \( \forall \) are referred to as guarded recursive types. Combining this with guarded recursion [Nakano 2000] in the form of a fixed point operator of type \( (\forall A \rightarrow A) \rightarrow A \) 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.

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 \( \text{next} \) similar to \( \forall \) used to express that a formula should be true one step from now. It also has an operation \( \text{finally} \) 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 \( \text{until} \) operator \( \phi \mathbf{U} \psi \) of LTL, which expresses that \( \phi \) holds now and for some more steps, after which \( \psi \) becomes true. Using this operator, we can encode the finally operator \( \phi \mathbf{tt} \mathbf{U} \phi \) stating that \( \phi \) will eventually become true. In programming terms, the until operator is the inductive type given by constructors

\[
\text{now} : A \rightarrow A \mathbf{U} B \quad \text{wait} : A \rightarrow \text{next}(A \mathbf{U} B) \rightarrow A \mathbf{U} B.
\]

and the fact that it is inductive should imply a termination property similarly to that of LTL: Elements of type \( A \mathbf{U} B \) will eventually produce a \( B \) after at most finitely many \( A \)-s, and similarly for elements of type \( \phi \mathbf{B} \). 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 \( \text{next} \) and
▷ in such a system breaks the termination guarantee of the \( \mathcal{U} \) type. For example, if \( a : A \), the fixed point of \( \text{wait} a : \bigcirc(A \ \mathcal{U} \ B) \rightarrow A \ \mathcal{U} \ B \) 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 \( X \equiv B + A \times \triangleright X \), 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 \( \text{Ev}(A) \) of possibly occurring events of type \( A \) can be encoded as the unique solution to \( \text{Ev}(A) \equiv A + \triangleright \text{Ev}(A) \). Using \( \bigcirc \), the type \( \bigtriangleup A \) 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 \( \bigtriangleup A \) into \( \text{Ev}(A) \). The lack of an inclusion from \( \triangleright \) to \( \bigcirc \) means that there is no inclusion \( \triangleright \bigtriangleup A \rightarrow \bigtriangleup A \) to take a fixed point of to construct a diverging element of \( \bigtriangleup A \).

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 et al. 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 \( \bigtriangledown \) and \( \bigtriangledown \bigcirc \) for \( \triangleright \) and \( \bigcirc \) respectively, and the inclusion of \( \bigcirc \) into \( \triangleright \) is defined by allowing \( \bigtriangledown \bigcirc \) to eliminate also \( \bigcirc \). We think of \( \bigtriangledown \bigcirc \) and \( \bigtriangledown \bigtriangledown \bigcirc \) as a separation in time in judgements: Variables to the left of \( \bigtriangledown \bigcirc \) or \( \bigtriangledown \bigtriangledown \bigcirc \) are available one time step before those to the right. The token \( \bigtriangledown \bigtriangledown \bigcirc \) 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 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.

These results are proved (section 5) using an interpretation of types as sets of values indexed by three parameters, including an ordinal \( v \). For finite \( v \), this index should be thought of as a form of step-indexing: The interpretation of \( A \) at \( v \) in this case describes the behaviour of terms up to
Fig. 1. Grammars for types, stable types and limit types. In typing rules, only closed types (no free α) are considered.

\[
A, B ::= \alpha | 1 | \text{Nat} | A \times B | A + B | A \rightarrow B | \square A | \Diamond A | \triangleright A | \text{Fix } \alpha.A
\]

Types

\[
S, S' ::= 1 | \text{Nat} | \square A | S \times S' | S + S'
\]

Stable types

\[
L, L' ::= \alpha | 1 | \text{Nat} | \triangleright A | \square L | \Diamond L | L \times L' | L + L' | A \rightarrow L | \text{Fix } \alpha.L
\]

Limit types

Fig. 2. Well-formed contexts

the first ν evaluation steps. In our model, however, ν runs all the way to ω · 2. The interpretation at higher ν, in particular the limit ordinal ω describes global behaviour of programs.

The distinction between ⊲ and □ can be seen in the model. At successor ordinals ν + 1, the interpretation of ⊲ A and □ A are both defined in terms of the interpretation of A at ν in a step-indexed fashion [Birkedal et al. 2011], but at limit ordinals ν, the interpretation of ⊲ A is the intersection of the interpretations at ν ′ < ν, whereas □ A is interpreted using the interpretation of A at ν. This interpretation of ⊲ A is needed to interpret fixed points, and the interpretation of □ A ensures that the interpretation of A 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).

2 LIVELY RATT

Lively RaTT is an extension of Simply RaTT [Bahr et al. 2019], a Fitch-style modal language for reactive programming. The type system of Simply RaTT guarantees 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. Although we do not extend the results on space leaks proved for Simply RaTT to Lively RaTT, we do maintain the restrictions on the language known to be necessary for these. 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 □ expresses delay of data by one time step and has introduction and elimination as follows (ignoring ⊲ for the moment).

\[
\frac{\Gamma, \triangleleft \circlearrowleft t : A}{\Gamma \vdash \text{delay } t : \Diamond A}
\]

\[
\frac{\Gamma \vdash t : \Diamond A}{\Gamma, \circlearrowleft \text{token-free}(\Gamma')} \quad \frac{\Gamma \vdash t : \Diamond A}{\Gamma, \circlearrowleft, \Gamma' \vdash \text{adv } t : A}
\]

The token ◀ 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 states that t has type A one time step after Γ, and thus delay t has type □ 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...
Diamonds are not forever

\[ \lambda f . \lambda x . \text{delay}((\text{adv} f)(\text{adv} x)) : \bigcirc (A \rightarrow B) \rightarrow \bigcirc A \rightarrow \bigcirc B \]  (1)

In the elimination rule, the premise \( \text{token-free}(\Gamma') \) states that \( \Gamma' \) does not contain tokens like \( \bigcirc \). This prevents \( \text{adv} t \) from being transported further into the future, a common source of space leaks. Similarly, variables can not be introduced over tokens.

Unlike Simply RaTT, Lively RaTT has two modalities for time delays: \( \bigcirc \) and \( \triangleright \). Both correspond to a time step in the execution of reactive programs, but in addition \( \triangleright \) corresponds to a time step in the sense of guarded recursion. Consequently, the \( \bigcirc \) token is stronger than \( \bigcirc \); Both can be used to advance time, but \( \bigcirc \) can also be used to unfold fixed points. We capture this extra strength in a reflexive ordering generated by \( \bigcirc \leq \triangleright \) on delay modalities, and allowing \( \bigcirc_{m'} \) to eliminate

\[ \text{Fig. 3. Typing rules. Here } m, m' \text{ ranges over the set } \{\bigcirc, \triangleright\} \text{ of time modalities ordered by } \bigcirc \leq \triangleright. \text{ In all rules, all contexts are assumed well-formed.} \]
modality \( m \) if \( m \leq m' \). This induces an inclusion

\[
\text{embed} = \lambda x.\text{delay}\,(\text{adv}\,x) : \square A \to \triangledown A
\]

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 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 \( \bigcirc \) and \( \triangledown \) can only appear to the right of a \( \sharp \).

The rules for well-formed contexts can be found in Figure 5. The rules for well-formed contexts can be found in Figure 2.

The token \( \# \) is associated with the modality \( \Box \). Data of type \( \Box A \) 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 \( \Box \) which ensures that box \( t \) can not contain free dynamic variables (i.e. variables to the right of \( \#$\)), and in the elimination rule allowing \( \Gamma \vdash t : \Box A \) to be eliminated in a context \( \Gamma, \# \), \( \Gamma' \) also when \( \Gamma' \) contains \( \bigcirc \) or \( \triangledown \).

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 using the constructions progress and promote. Note in particular that function types are not stable since closures can contain time-dependent data.

Elements of type \( \Box A \) can also be constructed as guarded recursive fixed points. These are particularly useful for programming with guarded recursive types, i.e., types of the form \( \text{Fix } \alpha.A \) satisfying the type isomorphism \( \text{Fix } \alpha.A \cong A[\triangledown(\text{Fix } \alpha.A)/\alpha] \). Note that there is no restriction on \( A \), which can in principle contain also negative occurrences of \( \alpha \), although we shall not be using that in this paper. The basic FRP types of streams and events can be encoded as guarded recursive types

\[
\text{Str}(A) \overset{\text{def}}{=} \text{Fix } \alpha.A \times \alpha \quad \quad \text{Ev}(A) \overset{\text{def}}{=} \text{Fix } \alpha.A + \alpha
\]

The fixed point combinator as defined by Nakano [2000] is simply a term of type \((\triangledown A \rightarrow A) \rightarrow A\). 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 \( \Box A \). In Simply RaTT, the typing rule for fixed points states that \( \Gamma \vdash \text{fix } x.t : \Box A \) if \( \Gamma, \# : \triangledown A \vdash t : A \). In Lively RaTT this is too restrictive, since there can be multiple \( \bigcirc \) in a context and we need access to the recursion variable \( x \) also under arbitrary many of these. The premise of the rule is therefore \( \Gamma, x : \Box\triangledown A, \# \vdash t : A \), which gives a more general fixed point rule.

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

\[
\text{map} : \Box (A \rightarrow B) \rightarrow \Box (\text{Str } A \rightarrow \text{Str } B)
\]

\[
\text{map} = \lambda f.\text{fix } m.\lambda a::as.\left(\text{unbox } f\right) a::\text{delay }((\text{adv } (\text{unbox } m)) (\text{adv } as))
\]

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 \( \Box (A \rightarrow B) \) 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 \( \Gamma \) independent of which tokens are in \( \Gamma \). The second is the until-type of LTL, to be thought of as the inductive solution to \( A \mathcal{U} B \cong B + A \times (A \mathcal{U} B) \). 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 \(A \times \Diamond (A \cup B)\) should recurse one time step from now on the advanced element of type \(A \cup B\). To avoid space leaks, the recursors should be stable, i.e., not depend on dynamic data. Thus eliminating from \(A \cup B\) into a type \(C\) requires recursors of type \(\Box(B \to C)\) and \(\Box(A \to \Diamond (A \cup B) \to \Diamond C \to C)\).

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 \(\boxcheck\) and \(\boxtimes\). 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 Diamonds

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

\[
\text{loopEvent} : \Box \text{Ev} A
\]

\[\text{loopEvent} = \text{fix } e . \text{into} (\text{in}_2 \text{ (unbox } e))\]

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

\[
\begin{align*}
\text{now}_\Diamond : A & \to \Diamond A \\
\text{now}_\Diamond a & = \text{now } a \\
\text{wait}_\Diamond : \Diamond A & \to \Diamond A \\
\text{wait}_\Diamond e & = \text{wait } e \text{ into } (\text{in}_2 \text{ (unbox } e))
\end{align*}
\]

Here the \(\text{now}_\Diamond\) and \(\text{now}_\text{Ev}\) maps are like the \(\text{return}\) map from a monad. Both Ev and \(\Diamond\) further admits a map reminiscent of the \(\text{bind}\) map. For Ev this is given by:

\[
\begin{align*}
\text{bind}_\text{Ev} : & \Box (A \to \text{Ev } B) \to \Box (\text{Ev } A \to \text{Ev } B) \\
\text{bind}_\text{Ev} f & = (\lambda f . (\text{fix } b . \lambda e . \text{case } e \text{ of } \text{now}_\text{Ev} a \cdot (\text{unbox } f) a \\
& \quad \text{wait}_\text{Ev} e . \text{wait}_\text{Ev} (\text{unbox } b \ominus e)))
\end{align*}
\]

where \(\ominus\) 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 \(m_i \in \{\ominus, \ominus\}\) with \(m_1, m_2 \leq m_3\). To see that \(\text{bind}_\text{Ev}\) 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 \(e : \text{Ev } A\) and \(b : \Box (\text{Ev } A \to \text{Ev } B)\). It then follows that unbox \(b : \ominus (\text{Ev } A \to \text{Ev } B)\) and thus, by a delayed function application, \((\text{unbox } b) \ominus e : \ominus (\text{Ev } B)\). This is then wrapped in \(\text{wait}_\text{Ev}\) to produce an element of Ev as needed. Note the requirement for the map \(f : A \to \text{Ev } B\) to be stable. This is because it might be applied in the future.

For \(\Diamond\), we define the map

\[
\begin{align*}
\text{bind}_\Diamond & : \Box (A \to \Diamond B) \to \Box (\Diamond A \to \Diamond B) \\
\text{bind}_\Diamond f & = (\lambda \text{dia}. \ Rec_{\text{FL}} (a \cdot (\text{unbox } f) a, u \ w \ d \cdot \text{wait}_\Diamond d, \text{dia}))
\end{align*}
\]

where again \(f\) must be stable. To see that \(\text{bind}_\Diamond\) 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 \(u : 1, w : \Diamond (A \cup B)\) and \(d : \Diamond \Diamond B\), hence also \(\text{wait}_\Diamond d : \Diamond B\) as required.
We will also use sugared syntax for recursive definitions, writing e.g. the above definition of \( \text{bind}_{Ev} \) as

\[
\text{bind}_{Ev} : \square (A \rightarrow \text{Ev} B) \rightarrow \square (\text{Ev} A \rightarrow \text{Ev} B)
\]

\[
\text{bind}_{Ev} f \# (\text{now}_{Ev} a) = (\text{unbox} f) a
\]

\[
\text{bind}_{Ev} f \# (\text{wait}_{Ev} e) = \text{wait}_{Ev} (\text{unbox} (\text{bind}_{Ev} f) \otimes e)
\]

The lock separates the variables into those received before and after \( \text{fix} \), and since the two cases define \( \text{bind}_{Ev} f \) by guarded recursion, this should be considered an atomic subexpression with type \( \square \triangleright (\text{Ev} A \rightarrow \text{Ev} B) \).

Similarly, the definition of \( \text{bind}_{\Diamond} \) can be written in the sugared syntax as

\[
\text{bind}_{\Diamond} : \square (A \rightarrow \Diamond B) \rightarrow \square (\Diamond A \rightarrow \Diamond B)
\]

\[
\text{bind}_{\Diamond} f = \text{box} \text{bind}_{\Diamond} '
\]

where \( \text{bind}_{\Diamond} ' : \Diamond A \rightarrow \Diamond B \)

\[
\text{bind}_{\Diamond} ' (\text{now}_{\Diamond} a) = (\text{unbox} f) a
\]

\[
\text{bind}_{\Diamond} ' (\text{wait}_{\Diamond} d) = \text{wait}_{\Diamond} (\text{bind}_{\Diamond} ' d)
\]

Here, the two cases of the recursive definition of \( \text{bind}_{\Diamond} ' \) are written as pattern matching syntax. In the second case the subterm \( \text{bind}_{\Diamond} ' d \) represents the recursive call and should therefore be read as having type \( \Diamond \Diamond B \). To elaborate such definitions back into \( \mathcal{U} \)-recursion, replace calls such as \( \text{bind}_{\Diamond} ' d \) 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.

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

\[
\text{diaInclusion} : \square (\Diamond A \rightarrow \text{Ev} A)
\]

\[
\text{diaInclusion} = \text{box} \text{diaInclusion}'
\]

where \( \text{diaInclusion}' : \Diamond A \rightarrow \text{Ev} A \)

\[
\text{diaInclusion}' (\text{now}_{\Diamond} a) = \text{now}_{Ev} a
\]

\[
\text{diaInclusion}' (\text{wait}_{\Diamond} d) = \text{wait}_{Ev} (\text{embed} (\text{diaInclusion}' d))
\]

This map makes crucial use of the fact that \( \Diamond \) is a sub-modality of \( \triangleright \) in the call to \( \text{embed} \), as defined in Equation 2.

A further consequence of the sub-modality relation is that non-terminating events “overrule” terminating events. Consider \( \text{Ev} \) containing a \( \Diamond \):

\[
\text{diamondEvent} : \square (\text{Ev} \Diamond A \rightarrow \text{Ev} A)
\]

\[
\text{diamondEvent} = \text{bind}_{Ev} \text{diaInclusion}
\]

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

There is in general no inclusion the from \( \text{Ev} \) 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
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:

\[ 0 \circ = \text{now} \circ \langle \rangle \]
\[ \text{suc} \circ n = \text{wait} \circ n \]
Any temporal natural number can be imported into the future by means of $U$-recursion.

$$
\begin{align*}
\text{import} : \text{Nat}_\Box &\to \Box \text{Nat}_\Box \\
\text{import} 0_\Box &\quad = \text{delay} (0_\Box) \\
\text{import} (\text{suc}_\Box n) &\quad = \text{delay} (\text{suc}_\Box (\text{adv} (\text{import} n)))
\end{align*}
$$

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

$$
\begin{align*}
timer : \text{Nat} &\to \text{Nat}_\Box \\
timer 0 &\quad = \text{delay} 0_\Box \\
timer (\text{suc} n) &\quad = \text{suc}_\Box (\text{import} (\text{timer} n))
\end{align*}
$$

Intuitively speaking, given a natural number $n$, $\text{timer} n$ 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.

$$
\begin{align*}
\text{buffer} : A \text{ stable} &\Rightarrow \Box (\text{Nat}_\Box \to A \to \Diamond A) \\
\text{buffer} &= \text{box} \text{ buffer}' \\
\text{where} \quad &\text{buffer}' : \text{Nat}_\Box \to A \to \Diamond A \\
&\quad \text{buffer}' 0_\Box a \quad = \text{now}_\Box a \\
&\quad \text{buffer}' (\text{suc}_\Box n) a \quad = \text{wait}_\Box ((\text{buffer}' n) \odot a)
\end{align*}
$$

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

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

where $\text{Req}$ and $\text{Resp}$ are the types of requests and responds, 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 $\text{Req} := \text{Nat} \times \text{String}$ and $\text{Resp} := \text{String} \times \text{Nat}$, and consider $\text{String}$ to be stable. The server is defined by guarded recursion:

$$
\begin{align*}
\text{rServer} : \Box (\text{Nat} \to \text{Server}) \\
\text{rServer} m &= \text{into} \langle \text{rServerFst}, \text{rServerSnd} \rangle \\
\text{where} \quad &\text{rServerFst} : \Diamond \text{Server} \\
&\quad \text{rServerFst} = (\text{unbox} \text{ rServer}) \odot m \\
&\quad \text{rServerSnd} : (\text{Nat} \times \text{String}) \to (\Diamond (\text{String} \times \text{Nat}) \times \Diamond \text{Server}) \\
&\quad \text{rServerSnd} \langle n, s \rangle = ((\text{unbox} \text{ buffer}) (\text{timer} n) \langle s, m \rangle, (\text{unbox} \text{ rServer}) \odot (\text{suc} m))
\end{align*}
$$

The server can be run and initialized with $0$:

$$
\begin{align*}
\text{rServerRun} : &\Box \text{ Server} \\
\text{rServerRun} &= \text{box} ((\text{unbox} \text{ rServer}) 0)
\end{align*}
$$

### 3.2 Fair streams

A stream of type $\text{Str}(A + B)$ 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:
\(altStr : \Box (\text{Str} A \to \text{Str} B \to \text{Str} (A + B))\)
\(altStr \# (a :: \text{delay} (a' :: as)) (b :: \text{delay} (b' :: bs)) =\)
\[\text{in}_2 b :: \text{delay} (\text{in}_1 a' :: (\text{unbox} altStr \odot as \oplus bs))\]

Here we also allow pattern matching with the delay introduction form, which – as usual – translates to an application of the corresponding elimination form adv in the calculus. That is, a pattern delay \(x\) translates to a variable pattern \(y\), if we replace all occurrences of \(x\) with \(\text{adv} y\).

The following function inhabits the same type, but it only draws elements from the first stream, dropping the second stream altogether:
\(\text{dropSnd} : \Box (\text{Str} A \to \text{Str} B \to \text{Str} (A + B))\)
\(\text{dropSnd} \# as bs = \text{unbox} (\text{map} (\text{box} \text{in}_1)) as\)

Following the work by Cave et al. [2014], we can refine the type \(\text{Str}(A + B)\) to a type \(\text{Fair}(A, B)\), whose inhabitants will produce in each step a value of type \(A\) or of type \(B\), but they do so in a fair manner:
\(\text{Fair}(A, B) = \text{Fix} \alpha. \text{U} (B \times \triangleright(B \text{U} (A \times \alpha)))\)

A term of type \(\text{Fair}(A, B)\) 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 \(\text{dropSnd}\) to produce a fair stream of type \(\text{Fair}(A, B)\). On the other hand, we can re-implement \(altStr\) to produce a fair stream as follows:
\(altFair : \Box (\text{Str} A \to \text{Str} B \to \text{Fair} (A, B))\)
\(altFair \# (a :: \text{delay} (a' :: as)) (b :: \text{delay} (b' :: bs)) =\)
\[\text{into} (\text{now} \langle b, \text{delay} (\text{now} \langle a', \text{unbox} altFair \odot as \oplus bs))\rangle)\]

To simplify programming with fair streams we define shortcut constructors for the type \(\text{Fair}(A, B)\). To this end we define the following variant of the type \(\text{Fair}(A, B)\):
\(\text{Fair'}(B, A) = B \text{U} (A \times \triangleright\text{Fair}(A, B))\)

We now have that \(\text{Fair}(A, B)\) unfolds to \(A \text{U} (B \times \triangleright\text{Fair'}(B, A))\) and thus the two types \(\text{Fair}(A, B)\) and \(\text{Fair'}(A, B)\) are isomorphic. Fair streams are constructed by either staying with the first type \(A\) or switching to the second type \(B\).
\(\text{stay} : A \to \bigcirc \text{Fair} (A, B) \to \text{Fair} (A, B)\)
\(\text{stay}_{a} d = \text{into} (\text{wait} a d)\)
\(\text{switch} : B \to \triangleright\text{Fair'} (B, A) \to \text{Fair} (A, B)\)
\(\text{switch}_{b} d = \text{into} (\text{now} \langle b, d \rangle)\)

From the types 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. The \(altFair\) function can thus be implemented more concisely as follows:
\(altFair : \Box (\text{Str} A \to \text{Str} B \to \text{Fair} (A, B))\)
\(altFair \# (a :: \text{delay} (a' :: as)) (b :: \text{delay} (b' :: bs)) =\)
\[\text{switch} b (\text{delay} (\text{switch'} a' (\text{unbox} altFair \odot as \oplus bs)))\]

The fair stream type \(\text{Fair}(A, B)\) can be considered a special case of the stream type \(\text{Str}(A + B)\) with additional liveness constraints. We can always forget these constraints by converting a fair stream into a normal stream:
runFair : □ (Fair (A, B) → Str (A + B))
runFair # = run1
  where run2 : Fair’ (B, A) → Str (A + B)
  run2 (stay’ b d) = in2 b :: embed (run2 d)
  run2 (switch’ a d) = in1 a :: unbox runFair ⊙ d
run1 : Fair (A, B) → Str (A + B)
run1 (stay a d) = in1 a :: embed (run1 d)
run1 (switch b d) = in2 b :: delay (run2 (adv d))

The function runFair is defined by guarded recursion with two nested ‘U-recursions on the
two nested ‘U-types that make up the fair stream type. Note that the two recursive calls run1 d
and run2 d produce a delayed stream of type ⊙(Str(A + B)). Therefore, we have to use embed to
convert them to type □(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:

sch : limit A, limit B ⇒ □ (Nat → Str A → Str B → Fair (A, B))
sch # n as bs = until (timer n) n as bs
  where until : Nat ⊙ → Nat → Str A → Str B → Fair (A, B)
  until (suc₀ n) m (a :: as) (b :: bs) = stay a (until n ⊙ m ⊗ as ⊗ bs))
  until 0 ⊙ m (a :: delay (a’ :: as)) (b :: delay (b’ :: bs))
    = switch b (delay (switch’ a’ (unbox sch ⊙ m + 1 ⊗ as ⊗ bs))

In particular unbox sch 0 as bs produces a fair stream of the following form:

    B A A B A A A B A A A A B A A A A A A B

The fair scheduler is implemented by guarded recursion with a nested ‘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 ⊙, 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 Str(A) and 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 until n of type ⊙(Nat → Str(A) → Str(B) → Fair(A, B)) to both as and bs, which are of type
▷Str(A) and ▷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 ⊙(Str(A) → 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.

, Vol. 1, No. 1, Article . Publication date: March 2020.
### Call-by-value $\lambda$-calculus:

| $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ |
|-------------|-------------|-------------|-------------|-------------|
| $v$ | $v$ | $t \downarrow v$ | $t' \downarrow v'$ | $t \downarrow \langle v_1, v_2 \rangle$ $i \in \{1, 2\}$ | $t \downarrow v$ $i \in \{1, 2\}$ |
| $\langle t, t' \rangle$ | $\langle v, v' \rangle$ | $\pi_i(t) \downarrow v_i$ | $\text{in}_i(t) \downarrow \text{in}_i(v)$ |

- $t \downarrow \text{in}_i(v)$ $t_i[v/x] \downarrow v_i$ $i \in \{1, 2\}$
- $t \downarrow \lambda x. s$ $t' \downarrow v$ $s[v/x] \downarrow v'$
- $t \downarrow v$
- $\text{unbox } t \downarrow v$
- $t \downarrow \text{delay } t'$
- $t' \downarrow v$

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

| $\downarrow$ | $\downarrow$ | $\downarrow$ | $\downarrow$ |
|-------------|-------------|-------------|-------------|
| $\text{adv } t \downarrow v$ | $\text{promote } t \downarrow v$ | $\text{progress } t \downarrow v$ | $\text{unbox } t \downarrow v$

- $t \downarrow v$
- $\text{now } t \downarrow \text{now } v$
- $t_1 \downarrow v_1$ $t_2 \downarrow v_2$ $u \downarrow \text{now } v$ $s[v/x] \downarrow \text{w}$
- $\text{rec}_\mathcal{U}(x.s, y.t, u) \downarrow \text{w}$
- $\text{rec}_\mathcal{U}(x.s, y.z.t, u) \downarrow \text{w}$
- $\text{rec}_\mathcal{U}(x.s, y.z.t, u) \downarrow \text{w}$

### 4.1 Evaluation semantics

The evaluation semantics is presented as a big-step operational semantics in Figure 4 and describes how a term $t$ evaluates to a value $v$ in the current time instant, denoted $t \downarrow v$. The grammar below describes which terms of the calculus are considered values:

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

Here we also include input locations $l$, which provide the interface for interaction with external inputs, but defer further discussion of input locations until section 4.3.

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 $\Box$, $\Diamond$, 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. Values of these modalities are of the form $\text{box } t$ and $\text{delay } t$ consisting of unevaluated terms $t$, which are only evaluated once they are consumed by $\text{unbox}$ and $\text{adv}$, respectively. Values of type $\Box A$ represent time invariant computations that can be invoked at any time to produce values of type $A$. Similarly, values of type $\Diamond A$ and $\triangleright A$, represent computations
that may be executed in the next time step to produce a value of type \( A \). These computations are suspended, since the evaluation semantics describes computations at the current time instant.

The operational semantics of the guarded fixed point combinator \( \text{fix} \) closely follows the intuition provided by its type: The fixed point \( \text{fix} x.t \) is unfolded by delaying it into the future and substituting it for \( x \) in \( t \). However, since \( \text{fix} x.t \) 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 \( \mu \alpha. B + (A \times \Box \alpha) \), is that each recursive call \( \text{rec}_\mathcal{U}(\ldots) \) is shifted one time step into the future via delay. 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 \( t \xrightarrow{\nu} t' \) indicates the passage of one time step during which the stream program \( t \) transitions to the program \( t' \) and emits the output \( \nu \). We give three separate step semantics for stream, until, and fair stream types, denoted \( \xrightarrow{\text{Str}} \), \( \xrightarrow{\mathcal{U}} \), and \( \xrightarrow{\text{F}} \), respectively. In addition, we state our metatheoretic results: \( \xrightarrow{\text{Str}} \) is productive, \( \xrightarrow{\mathcal{U}} \) is guaranteed to terminate, and \( \xrightarrow{\text{F}} \) indeed produces a fair stream. But we defer the proofs of these results to section 5.

A closed term \( t \) of type \( \Box \text{Str}(A) \) is meant to produce an infinite stream \( \nu_1, \nu_2, \nu_3, \ldots \) of values of type \( A \) in a step-by-step fashion:

\[
\text{unbox } t \xrightarrow{\nu_1} \text{Str } t_1 \xrightarrow{\nu_2} \text{Str } t_2 \xrightarrow{\nu_3} \text{Str } \ldots
\]

Each step \( t_i \xrightarrow{\nu_i} \text{Str } t_{i+1} \) proceeds by first evaluating \( t \) to the value \( \nu_i : w \), i.e., the head \( \nu_i : A \) and the tail \( w : \triangleright \text{Str}(A) \) of the stream. Computation may then proceed in the next time step with the term \( t_{i+1} = \text{adv } w \). This application of the \( \text{adv} \) combinator signifies the shift to the next time step.

We can show that a term of type \( \Box \text{Str}(A) \) indeed produces such an infinite sequences 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:

\[
U, V ::= 1 \mid \text{Nat} \mid U \times V \mid U + V
\]

### Theorem 4.1  (productivity)

If \( \vdash t : \Box \text{Str}(A) \), then there is an infinite sequence of reduction steps

\[
\text{unbox } t \xrightarrow{\nu_1} \text{Str } t_1 \xrightarrow{\nu_2} \text{Str } t_2 \xrightarrow{\nu_3} \text{Str } \ldots
\]

Moreover, if \( A \) is a value type, then \( \vdash \nu_i : A \) for all \( i \geq 1 \).
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 $\vdash v_1 : A$.

The operational semantics for until stream 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 $\kappa$.

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

$$\text{unbox } t \Rightarrow u \, t_1 \Rightarrow u \, t_2 \Rightarrow u \, \ldots \Rightarrow u \text{ HALT}$$

Moreover, if $A$ and $B$ are value types, then $\vdash v_1 : A$ for all $0 < i < n$, and $\vdash v_n : B$.

The computation performed by a fair stream of type $\Box \text{Fair}(A, B)$ requires a bit of additional bookkeeping: The state of the program is represented by a pair $(t; p)$ consisting of a term $t$ 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 $\vdash t : \Box \text{Fair}(A, B)$, then there is an infinite sequence of reduction steps

$$\langle \text{out (unbox } t) ; 1 \rangle \xmapsto{\text{in}_{p_1} \, v_1} \langle t_1 ; p_1 \rangle \xmapsto{\text{in}_{p_2} \, v_2} \langle t_2 ; p_2 \rangle \xmapsto{\text{in}_{p_3} \, v_3} \ldots$$

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 $\vdash \text{in}_{p_i} \, v_i : A + B$ for all $i \geq 1$.

As a special case of the fair stream type we obtain the type $\text{Live}(A) = \text{Fair}(1, A)$. 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 allow for interaction with external input, we extend the language with **input locations** $l$, which represent the promise of an input in the next time step. Note however that this does not change the surface language; in particular, we do not add a typing rule for locations. We only require that input locations are values and thus evaluate to themselves, i.e., $l \Downarrow l$.

The promise represented by an input location $l$ is redeemed by substituting a value of the form delay $s$ for $l$. We describe this action of delivering promised inputs into a term $t$ by applying an **input substitution** $\sigma$, denoted $t \sigma$. An input substitution $\sigma$ is a finite mapping from input locations to terms, and the term $t \sigma$ is obtained from $t$ by replacing all occurrences of an input location $l \in \text{dom}(\sigma)$ in $t$ by delay $\sigma(l)$ unless that occurrence is in the scope of box or fix. That is, in particular, we have the following equalities:

$$(\text{box } t) \sigma = \text{box } t \quad (\text{fix } x . t) \sigma = \text{fix } x . t \quad l \sigma = \begin{cases} \text{delay } \sigma(l) & \text{if } l \in \text{dom}(\sigma) \\ l & \text{if } l \notin \text{dom}(\sigma) \end{cases}$$

In the following we pick a fixed input location $l^*$ and define for each value $v$ the singleton input substitution $\sigma_v$ that maps $l^*$ to the value $v :: l^*$. That is, the promise $l^*$ is replaced by a stream with head $v$ and the promise $l^*$ of a tail. Each time such an input substitution $\sigma_v$ is applied, we thus provide the next element of an input stream.
Fig. 6. Reactive step semantics for streams, until types and fair streams.

For streams, the reactive step semantics describes computation steps of the form \( t \xrightarrow{\text{Str}} t' \) that start in a term \( t \), read an input value \( v \), produce and output value \( v' \), and then transition to a term \( t' \). Using the above notion of input substitutions, \( t \xrightarrow{\text{Str}} t' \) is simply defined as the computation step \( t\sigma_v \xrightarrow{\text{Str}} t' \).

Given a term \( t \) of type \( \square(\text{Str}(A) \rightarrow \text{Str}(B)) \), and a sequence \( v_1, v_2, \ldots \) of values of type \( A \), there is an infinite sequence of reduction steps

\[
\text{unbox } t(\text{adv } l^*) \xrightarrow{\text{Str}} t_1 \xrightarrow{\text{Str}} t_2 \xrightarrow{\text{Str}} \cdots
\]

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 unbox \( t \) of type \( \text{Str}(A) \rightarrow \text{Str}(B) \) the argument \( \text{adv } l^* \). While \( \text{adv } l^* \) is not a well-typed term, it has the type \( \text{Str}(A) \) semantically, in the sense that \( \text{adv } l^* \) is a term in the logical relation \( [\text{Str}(A)] \) that we construct in section 5.

The first step of the reduction sequence applies the input substitution \( \sigma_{v_i} \) and thus turns the argument term \( \text{adv } l^* \) into \( \text{adv } (\text{delay}(v_1 :: l^*)) \), which evaluates to \( v_1 :: l^* \); i.e., the first element of the input stream together with a promise for the remainder of the input stream.

The step semantics for \( \mathcal{U} \)-types and fair types can be generalised to a reactive step semantics in the same fashion. Figure 6 shows the formal definition of the reactive variant of the step semantics of Figure 5.

The type system of Lively RaTT ensures that future inputs indicated by input locations are not prematurely evaluated, i.e., the operational semantics will never try to evaluate a term \( \text{adv } l \). We shall show in section 5 that this will not happen, and as a consequence Lively RaTT respects the principle of causality:

**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 \( \vdash t : \square(\text{Str}(A) \rightarrow \text{Str}(B)) \), then there is an infinite sequence of reduction steps

\[
\text{unbox } t(\text{adv } l^*) \xrightarrow{\text{Str}} t_1 \xrightarrow{\text{Str}} t_2 \xrightarrow{\text{Str}} \cdots
\]

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

(ii) If \( \vdash t : \square(\text{Str}(A) \rightarrow B \mathcal{U} C) \), then there is a finite sequence of reduction steps

\[
\text{unbox } t(\text{adv } l^*) \xrightarrow{\mathcal{U}} t_1 \xrightarrow{\mathcal{U}} t_2 \xrightarrow{\mathcal{U}} \cdots \xrightarrow{H A L T}
\]

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 \( \vdash t : \square(\text{Str}(A) \rightarrow \text{Fair}(B, C)) \), then there is an infinite sequence of reduction steps

\[
\langle \text{out } (\text{unbox } t(\text{adv } l^*)); 1 \rangle \xrightarrow{\text{F}} \langle t_1 ; p_1 \rangle \xrightarrow{\text{F}} \langle t_2 ; p_2 \rangle \xrightarrow{\text{F}} \cdots
\]

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 \( \vdash \text{in } p_i v_i' : B + C \) for all \( i \geq 1 \).
Since the operational semantics is deterministic, in each step \( t_i \xrightarrow{\nu_{i+1}/\nu'_{i+1}}_{\text{Str}} t_{i+1} \) the resulting output \( \nu'_{i+1} \) and new state of the computation \( t_{i+1} \) are uniquely determined by the previous state \( t_i \) and the input \( \nu_{i+1} \). Thus, \( \nu_{i+1} \) and \( t_{i+1} \) are independent of future inputs \( \nu_j \) with \( j > i + 1 \). The same is true for the corresponding operational semantics of \( \mathcal{U} \)-types and fair streams.

5 METATHEOREY

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 \( \llbracket A \rrbracket_w \) 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 \( t \in \llbracket A \rrbracket_w \).

5.1 Worlds

To a first approximation, the worlds in our logical relation contain two ordinals \( \mu < \omega \) and \( \nu < \omega \cdot 2 \). Both of these 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 \( \llbracket \mathcal{U} A \rrbracket_{\mu, \nu} \) in terms of \( \llbracket A \rrbracket_{\mu, \nu} \) for strictly smaller \( \nu' \). Since unfolding Fix \( \alpha.A \) introduces a \( \triangleright \), we achieve that the step index decreases for guarded recursion types. For the inductive types, we define \( \llbracket A \mathcal{U} B \rrbracket_{\mu, \nu} \) in terms of \( \llbracket \Box (A \mathcal{U} B) \rrbracket_{\mu', \nu} \) where \( \mu' < \mu \). Thus, intuitively \( \mu \) gives an upper limit to the number of of the inductive type used in terms.

While this setup is sufficient for proving productivity, termination, and liveness properties, it is not enough to capture causality. To characterise causality, the logical relation also needs to account for what possible inputs a given term may receive. We achieve this by indexing the logical relation by a sequence of all future inputs. To this end we use the input substitutions \( \sigma \) introduced in section 4.3 to describe what input to expect at one particular point in time. An infinite sequence of input substitutions, called an input sequence and denoted by \( \overline{\sigma} \), describes the input that is received at each point in time in the future. In addition to infinite input sequences, we also allow the empty input sequence, denoted by \( \perp \). The empty input sequence is used to characterise terms that are unaffected by inputs.

As part of the Kripke structure of our logical relation we require that the logical relation be closed under supplying any finite prefix of an infinite input sequence. To capture this, we define a notion of maps between input sequences by the following rules:

\[
\begin{align*}
\varepsilon : \overline{\sigma} & \rightarrow \overline{\sigma} \\
\sigma' : \overline{\tau} & \rightarrow \overline{\tau}' \\
\sigma' \circ \sigma : \overline{\sigma; \tau} & \rightarrow \varepsilon ; \overline{\tau} \\
\sigma : \perp & \rightarrow \perp
\end{align*}
\]

Thus a map between non-empty input sequences is a input substitution that supplies a finite prefix of the domain input sequence replacing these by empty substitutions \( \varepsilon \) in the target. In particular, we have that \( \varepsilon \circ \sigma : \overline{\sigma; \tau} \rightarrow \varepsilon ; \overline{\tau}, \) i.e., after we apply the first input substitution \( \sigma \) of the sequence \( \overline{\sigma; \tau} \), we are left with only the input promises in the remainder of the sequence, namely \( \overline{\tau} \). Note that \( \varepsilon \circ \sigma = \sigma \), and we will in general not write the \( \varepsilon \).

As is standard for Kripke logical relations, our relation will be closed under moving to a bigger world.
5.2 Logical Relation

Our logical relation consists of two parts: A value relation $\mathcal{V}[A]_w$ that contains all values that semantically inhabit type $A$ at the world $w$, and a corresponding term relation $\mathcal{T}[A]_w$ containing terms. Given a world $w = (\mu, v, I)$ where $\mu \leq \omega, v < \omega \cdot 2$ and $I$ is either an infinite sequence $\sigma$ or the empty sequence $\bot$, we write $\mathcal{V}[A]_{\mu, v}$ and $\mathcal{T}[A]_{\mu, v}$ rather than $\mathcal{V}[A]_{(\mu, v, I)}$ and $\mathcal{T}[A]_{(\mu, v, I)}$. 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 $(\nu, |A|, \mu, e)$, where $|A|$ is the size of $A$ defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$|\sigma| = |\triangleright A| = |1| = |\text{Nat}| = 1$$

$$|A \times B| = |A + B| = |A \cup B| = |A \rightarrow B| = 1 + |A| + |B|$$

$$|\Box A| = |\bigcirc A| = |\text{Fix } \alpha.A| = 1 + |A|$$

Note that since $|\triangleright (\text{Fix } \alpha.A)| = |\sigma|$, the size of $A[\triangleright \text{Fix } \alpha.A/\alpha]$ is strictly smaller than that of $\text{Fix } \alpha.A$, which justifies the well-foundedness of recursive types. Note also that $\mathcal{V}[A \cup B]_{\mu, v}$ is defined in terms of $\mathcal{V}[\bigcirc (A \cup B)]_{\mu', v}$ for $\mu' < \mu$. To obtain well-foundedness, we would need $|\bigcirc (A \cup B)| \leq |A \cup B|$, which is not true. But this problem can be avoided by “inlining” the definition of $\mathcal{V}[\bigcirc (A \cup B)]_{\mu', v}$, which is defined in terms of $\mathcal{T}[A \cup B]_{\mu', v}$. For the sake of readability, we will keep the definitions as given.

The definition for $A \rightarrow B$ has two separate clauses depending on the input sequence $I$. In the case of the empty input sequence, the definition contains only lambda abstractions that can be applied in a future world and which are locked. Locked terms are the terms which are unaffected by input substitutions, i.e., $\forall \sigma.t = t\sigma$. Conceptually, any term in $\mathcal{V}[A]_{\mu, v}$ may not have any free locations and hence be unaffected by any input substitution, and for the function space, this needs to be explicitly required. The other clause for the function space contains only lambda abstractions that can be applied in a future world and under hence under any input substitution. This is to ensure closure under input substitutions.

The definition of $\Box A$ expresses the fact that all its inhabitants can be evaluated 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 $v$ is a limit ordinal. In the successor case, their elements are either delayed computation or future inputs that can be executed in the next time step. At that point, the next input has been substituted in, as presented by an input substitution of the form $\alpha : \sigma; \overline{\tau} \rightarrow e; \overline{\tau}$. The more general form of $\tau$ occurring in the definition is needed to ensure the Kripke structure as in Lemma 5.1 below. If $v$ is a limit ordinal, $\bigcirc A$ has the same interpretation except that the index $v$ is fixed. This is to ensure that inductive types have the correct behaviour at the limit. On the other hand, $\triangleright A$ is defined to be the intersection of the interpretation at all smaller $(v)$-indices, which forces $\triangleright A$ to be a limit type. This definition is needed for the interpretation of fixed points.

In the definition of $A \cup B$ we see the use of the $\mu$-index to give an upper bound of the number of unfoldings used in the elements of the logical relation. In particular, if $\mu = 0$, the relation contains only values of the form now $v$ whereas if $\mu > 0$, the relation also contain values of the form wait $u$ $w$ defined in terms of values from $\mathcal{V}[A \cup B]_{\mu', v}$ where $\mu' < \mu$.

Our value and term interpretation is closed w.r.t the Kripke structure on $v$ and $I$ and the value relation is upwards closed w.r.t $\mu$ for $\cup$-types.

Lemma 5.1 (Kripke Properties). Given $A, \mu, \mu', v, v', I, I'$ then we have

1. $v \subseteq v' \Rightarrow \mathcal{V}[A]_{\mu, v} \subseteq \mathcal{V}[A]_{\mu, v'}$  
2. $v \subseteq v' \Rightarrow \mathcal{T}[A]_{\mu, v} \subseteq \mathcal{T}[A]_{\mu, v'}$
\[ \mathcal{V}^n_{\mu, v} = \{ () \} , \]
\[ \mathcal{V}^n_{[\text{Nat}]}_{\mu, v} = \{ \text{suc}^n 0 \mid n \in \mathbb{N} \} , \]
\[ \mathcal{V}^n_{[A \times B]}_{\mu, v} = \{ (v_1, v_2) \mid v_1 \in \mathcal{V}^n_{[A]}_{\mu, v} \land v_2 \in \mathcal{V}^n_{[B]}_{\mu, v} \} , \]
\[ \mathcal{V}^n_{[A + B]}_{\mu, v} = \{ \text{in}_1 v \mid v \in \mathcal{V}^n_{[A]}_{\mu, v} \} \cup \{ \text{in}_2 v \mid v \in \mathcal{V}^n_{[B]}_{\mu, v} \} , \]
\[ \mathcal{V}^n_{[A \rightarrow B]}_{\mu, v} = \{ \lambda x.t \mid \text{locked}(t) \land \forall \nu' \leq \nu. \forall \nu \in \mathcal{V}^n_{[A]}_{\mu, v}.t[v/x] \in \mathcal{T}^n_{[B]}_{\mu, \nu'} \} , \]
\[ \mathcal{V}^n_{[\Box A]}_{\mu, v} = \{ v \mid \forall \nu \sigma : \overline{\sigma} \rightarrow \overline{\sigma}'. \forall \nu \in \mathcal{V}^n_{[A]}_{\mu, v}'.(t_\sigma)[v/x] \in \mathcal{T}^n_{[B]}_{\mu, \nu'} \} , \]
\[ \mathcal{V}^n_{[\Diamond A]}_{\mu, v} = \left[ \text{Loc} \cup \{ \text{delay } t \mid t \in \text{Terms} \} \right] \text{ if } \nu = 0 \]
\[ \left[ \{ v \mid \forall (\tau : \overline{\sigma} \rightarrow \overline{\sigma}). \text{adv}(v \tau) \in \mathcal{T}^n_{[A]}_{\mu, \nu'} \} \right] \text{ if } \nu = \nu' + 1 \]
\[ \left[ \{ v \mid \forall (\tau : \overline{\sigma} \rightarrow \overline{\sigma}). \text{adv}(v \tau) \in \mathcal{T}^n_{[A]}_{\mu, \nu'} \} \right] \text{ if } \nu \text{ limit ordinal} \]
\[ \mathcal{V}^n_{[\text{Fix } \alpha A]}_{\mu, v} = \left[ \text{into}(v) \mid v \in \mathcal{V}^n_{[A \triangleright \text{Fix } \alpha A]/[\alpha]}_{\mu, v} \right] , \]
\[ \mathcal{V}^n_{[A \; \mathcal{U} \; B]}_{\mu, v} = \{ \text{wait } v w \mid v \in \mathcal{V}^n_{[A]}_{\infty, v} \land \exists \mu' < \mu. w \in \mathcal{V}^n_{[\Diamond (A \; \mathcal{U} \; B)]}_{\mu', v} \} \cup \{ \text{now } v \mid v \in \mathcal{V}^n_{[B]}_{\infty, v} \} \]
\[ \mathcal{T}^n_{[\Box A]}_{\mu, v} = \{ t \mid \exists v.t \downarrow v \land v \in \mathcal{V}^n_{[A]}_{\mu, v} \} . \]

Fig. 7. Logical Relation.

(3) \( \forall \sigma : I \rightarrow I' . v \in \mathcal{V}^n_{[A]}_{\mu, v} \Rightarrow \nu \sigma \in \mathcal{V}^n_{[A]}_{\mu', v} \)
(4) \( \mu \leq \mu' \Rightarrow \mathcal{V}^n_{[A \; \mathcal{U} \; B]}_{\mu, v} \subseteq \mathcal{V}^n_{[A \; \mathcal{U} \; B]}_{\mu', v} \)

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

**Lemma 5.2** (Sub-modality). Given \( A, \mu, v \) and \( I \), then
\[ \mathcal{V}^n_{[\Diamond A]}_{\mu, v} \subseteq \mathcal{V}^n_{[\triangleright A]}_{\mu, v} \]

PROOF. Follows by transfinite induction on \( \nu \).

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 \( v \) is a limit ordinal, then
\[ \bigcap_{\nu' < \nu} \mathcal{V}^n_{[A]}_{\mu, v'} = \mathcal{V}^n_{[A]}_{\mu, v} \]
\[ \bigcap_{\nu' < \nu} \mathcal{T}^n_{[A]}_{\mu, v'} = \mathcal{T}^n_{[A]}_{\mu, v} \]
Lemma 5.1

Lemma 5.3

Lemma 5.2

section 4.3

\[ \sigma \], and the two sets are equal by definition. It suffices to show that if \( \sigma \), then \( A \in C[\Gamma]_\mu^\Gamma \) holds. The second equality then follows using determinism of the evaluation semantics.

\[ \sigma \]

Fig. 8. Context Relation

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 using determinism of the evaluation semantics.

\[ \sigma \]

In the special case where \( A \) is a limit type, \( \circ \) and \( \triangleright \) do in fact coincide:

Corollary 5.4 (Sub-modality at limit). Given \( A, \mu, \nu \) and \( I \) s.t. \( A \) limit, then

\[ \nu \Vdash [\circ A]_{\mu, \nu} = \nu \Vdash [\triangleright A]_{\mu, \nu} \]

PROOF. One inclusion always holds by Lemma 5.2, and the two sets are equal by definition except when \( \nu \) is a limit ordinal. In that case, by Lemma 5.3 it suffices to show that if \( \nu \in \nu \Vdash [\triangleright A]_{\mu, \nu} \) and \( \tau : \sigma \to \epsilon; \sigma \) then \( \text{adv}(\nu \tau) \in \tau [A]_{\mu, \nu} \), for all \( \nu' < \nu \), which follows from \( \nu \in \nu \Vdash [\triangleright A]_{\mu, \nu' + 1} \)

Finally, we obtain the soundness of the language by the following fundamental property of the logical relation \( \tau [A]_{\alpha, \nu} \).

Theorem 5.5 (Fundamental Property). If \( \Gamma \vdash t : A \) and \( \gamma \in C[\Gamma]_\mu^\Gamma \), then \( t\gamma \in \tau [A]_{\alpha, \nu} \).

Here \( C[\Gamma]_\mu^\Gamma \) refers to the logical relation for typing contexts defined in Figure 8. Note the case for \( \Gamma, \triangleright \), which captures the intuition that variables occurring before \( \triangleright \) arrive one time step before those to the right.

The theorem is proved by a lengthy but entirely standard induction on the typing relation \( \Gamma \vdash - : A \). Crucially, we use the fact that some substitutions and terms are locked and that semantically, \( \circ \) is a sub-modality of \( \triangleright \).

As an easy consequence of the fundamental property and the fact the empty substitution is an element of \( C[\#]_{\nu}^\Gamma \) for any non-empty input sequence \( \sigma \), we have the following property that we shall use to prove Lively RaTT’s operational properties:

Corollary 5.6 (Fundamental Property). If \( \# \vdash t : A \), then \( t \in \tau [A]_{\alpha, \nu} \) for all \( \nu \) and \( \sigma \).

5.3 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 \( \mu, \nu \), value type \( A \), and value \( \nu \), we have \( \nu \in \nu \Vdash [A]_{\mu, \nu} \iff \nu \vdash \mu : A \).

PROOF. By a straightforward induction on \( A \).
5.3.1 Productivity. We start with the productivity property of streams of type \( \text{Str}(A) \). Given a type A, we define the following set of terms

\[
T^\sigma_{\varphi}(A) = \left\{ t \mid t \sigma \in \mathcal{T}[\text{Str}(A)]^\sigma_{\text{io},v} \right\}
\]

Intuitively speaking, a term \( t \) in \( T^\sigma_{\varphi}(A) \) will produce a stream of A given the input sequence \( \varphi \). We formulate and prove the essence of the productivity property of such a stream as follows:

**Lemma 5.8** (productivity). Given \( t \in T^\sigma_{\varphi+1}(A) \), there are \( t' \) and \( v \) such that

\[
t \sigma \xrightarrow{v} \text{Str} t' \quad \text{and} \quad t' \in T^\varphi_{\varphi+1}(A), \quad v \in \mathcal{V}[A]^\varphi_{\text{io},v+1}
\]

**Proof.** Let \( t \sigma \in \mathcal{T}[\text{Str}(A)]^\sigma_{\text{io},v+1} \). Then \( t \sigma \Downarrow w \) and \( w \in \mathcal{V}[\text{Str}(A)]^\varphi_{\text{io},v+1} \). Hence, \( w = v :: v' \) with \( v \in \mathcal{V}[A]^\varphi_{\text{io},v+1} \) and \( v' \in \mathcal{V}[\text{Str}(A)]^\varphi_{\text{io},v+1} \). Given \( \sigma = \tau; \varphi \), we have that \( \tau : \varphi \rightarrow \varepsilon; \varphi \). Consequently, \( \text{adv} \ v' \tau \in \mathcal{T}[\text{Str}(A)]^\varphi_{\text{io},v} \), which means that \( \text{adv} \ v' \in T^\varphi_{\varphi+1}(A) \). Moreover, we have \( t \sigma \xrightarrow{v} \text{Str} \text{adv} v' \).

In each step of a stream computation, we consume the first element of the input sequence \( \varphi \), and count down by one on the index \( v \). For closed streams without any external input, we simply instantiate the above lemma with the sequence of empty inputs \( \varepsilon; \varepsilon; \ldots \), denoted \( \bar{\varphi} \).

**Proof of Theorem 4.1.** By Corollary 5.6 we have that unbox \( t \in T^\varphi_{\varphi}(A) \) for any \( \varphi \). Using Lemma 5.8, we can thus extend any finite reduction sequence

\[
\text{unbox} \ t \xrightarrow{\varphi} t_1 \xrightarrow{\varphi} t_2 \xrightarrow{\varphi} \cdots \xrightarrow{\varphi} t_n
\]

by an additional reduction step \( t_n \varepsilon \xrightarrow{\varphi} t_{n+1} \). Since \( t_n \varepsilon = t_n \) and \( \xrightarrow{\text{Str}} \) is deterministic, this uniquely defines the desired infinite reduction. Moreover, given that \( A \) is a value type, \( \vdash \ \nu_i : A \) follows for all \( i \geq 1 \) by Lemma 5.7.

5.3.2 Termination. Analogously to the set of terms \( T^\varphi_{\varphi}(A) \) for stream types, we define the following set \( U^\varphi_{\varphi}(A,B) \) for until types:

\[
U^\varphi_{\varphi}(A,B) = \left\{ t \mid t \sigma \in \mathcal{T}[A \ U B]^\varphi_{\text{io},v} \right\}
\]

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

**Lemma 5.9** (termination). Given \( t \in U^\varphi_{\varphi}(A,B) \), one of the following two statements holds:

(a) There are \( t' \), \( \mu' < \mu \), and \( v' \in \mathcal{V}[A]^\varphi_{\text{io},v} \) such that

\[
t \sigma \xrightarrow{v'} t' \quad \text{and} \quad \text{if} \ v > 0 \ \text{then} \ t' \in U^\varphi_{\varphi}(A,B)
\]

where \( v - 1 = v' \) if \( v = v' + 1 \) and otherwise \( v - 1 = v \).

(b) There is some \( v' \in \mathcal{V}[B]^\varphi_{\text{io},v} \) such that \( t \sigma \xrightarrow{v'} \text{HALT} \).

**Proof.** Let \( t \in U^\varphi_{\varphi}(A,B) \) for some \( \mu, v \). Then there is some \( w \in \mathcal{V}[A \ U B]^\varphi_{\text{io},v} \) with \( t \sigma \Downarrow w \). Hence, there are two cases for \( w \):

(a) \( w = \text{wait} v' \ w' \) with \( v' \in \mathcal{V}[A]^\varphi_{\text{io},v} \) and \( w' \in \mathcal{V}[\bigcirc(A \ U B)]^\varphi_{\mu',v} \) for some \( \mu' < \mu \). Hence, we have that \( t \sigma \xrightarrow{v'} \text{adv} w' \). Since \( \varphi \) is of the form \( \sigma'^*; \varphi' \), we know that \( \sigma' : \varphi \rightarrow \varepsilon; \varphi' \). Consequently, given that \( v > 0 \), we have \( \text{adv} (w' \sigma') \in T[A \ U B]^\varphi_{\mu',v-1} \), which means that \( \text{adv} w' \in U_{\mu',v-1}(A,B) \).
(b) \( w = \text{now} \ u' \) with \( u' \in \mathcal{V}[B]^{\overline{\sigma} \overline{\nu}} \). Then \( t\sigma \xrightarrow{\mathcal{U}} \text{HALT} \) follows immediately. \( \square \)

**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 unbox \( t \in U^{\overline{\sigma} \overline{\nu}}_{\omega, \omega}(A, B) \), and by Lemma 5.9 we can construct the desired sequence of reductions. Since the index \( \mu \) 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 \( u_i \) have the desired type given that \( A \) and \( B \) are value types. \( \square \)

### 5.3.3 Liveness

Recall that the step semantics of fair streams \( \xrightarrow{\mathcal{F}} \) works on pairs of the form \( \langle t; p \rangle \), 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 \( F^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B) \) of such pairs:

\[
F^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B) = \left\{ \langle t; 1 \rangle \mid t \in U^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B \times \triangleright \text{Fair}'(B, A)) \right\} \\
\cup \left\{ \langle t; 2 \rangle \mid t \in U^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(B, A \times \triangleright \text{Fair}(A, B)) \right\}
\]

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

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

**Lemma 5.10** (liveness). Given \( \langle t; p \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B) \), one of the following statements is true:

(a) there are \( t', \mu' < \mu \), and \( \nu' \in \mathcal{V}[A + B]^{\overline{\sigma} \overline{\nu}} \) such that

\[\langle t\sigma; p \rangle \xrightarrow{\mathcal{F}} \langle t'; p \rangle \quad \text{and} \quad \langle t'; p \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\mu', \nu'}(A, B) \quad \text{for all} \quad \nu' < \nu.\]

(b) there are \( t' \), and \( \nu' \in \mathcal{V}[A + B]^{\overline{\sigma} \overline{\nu}} \) such that

\[\langle t\sigma; p \rangle \xrightarrow{\mathcal{F}} \langle t'; 3 - p \rangle \quad \text{and} \quad \langle t'; 3 - p \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\nu}(A, B) \quad \text{for all} \quad \nu' < \nu.\]

**Proof.** Let \( B' = B \times \triangleright (B \mathcal{U} A') \) and \( A' = A \times \triangleright \text{Fair}(A, B) \). Let \( \langle t; p \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B) \). We only give the argument for \( p = 1 \) as the other case is analogous.

If \( p = 1 \), then \( t \in U^{\overline{\sigma} \overline{\nu}}_{\mu, \nu}(A, B') \). Hence, by Lemma 5.9, we have two cases:

(a) There is some \( \mu' < \mu \) such that \( t\sigma \xrightarrow{\mathcal{U}} t', v \in \mathcal{V}[A + B]^{\overline{\sigma} \overline{\nu}} \) and \( t' \in U^{\overline{\sigma} \overline{\nu}}_{\mu', \nu - 1}(A, B') \) given \( \nu > 0 \). Hence, \( \nu_1 \in \mathcal{V}[A + B]^{\overline{\sigma} \overline{\nu}} \). Moreover, by Lemma 5.1 we have that \( t' \in U^{\overline{\sigma} \overline{\nu}}_{\mu', \nu'}(A, B') \) for all \( \nu' < \nu \), which in turn implies that \( \langle t'; 1 \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\mu', \nu'}(A, B) \) for all \( \nu' < \nu \). Finally, by definition of \( \xrightarrow{\mathcal{F}} \), we have that \( \langle t\sigma; 1 \rangle \xrightarrow{\mathcal{F}} \langle t'; 1 \rangle \).

(b) We have \( t\sigma \xrightarrow{w} \text{HALT} \) with \( w \in \mathcal{V}[B]^{\overline{\sigma} \overline{\nu}} \). The latter implies that \( w = \langle v, u' \rangle \) with \( v \in \mathcal{V}[B]^{\overline{\sigma} \overline{\nu}} \) and \( u' \in \mathcal{V}[B \mathcal{U} A']^{\overline{\sigma} \overline{\nu}} \). Hence, \( \nu_2 \in \mathcal{V}[A + B]^{\overline{\sigma} \overline{\nu}} \). Since \( \overline{\sigma} \) is of the form \( \sigma'; \overline{\sigma} \), we have that \( \sigma'; \overline{\sigma} \rightarrow \epsilon; \overline{\sigma} \). Hence, \( \text{adv}(\nu' \sigma') \in \mathcal{T}[B \mathcal{U} A']^{\overline{\sigma} \overline{\nu}} \) for all \( \nu' < \nu \).

That is, we have that \( \langle t\sigma; 1 \rangle \xrightarrow{\mathcal{F}} \langle \text{adv} \nu'; 2 \rangle \) and \( \langle \text{adv} \nu'; 2 \rangle \in F^{\overline{\sigma} \overline{\nu}}_{\nu}(A, B) \) for all \( \nu' < \nu \). \( \square \)

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 \( \text{out}(\text{unbox } t) \in F_{\omega_1, \omega + n}(A, B) \) for any \( n \). Using Lemma 5.10, we can then show that we can extend any finite reduction sequence

\[
\langle \text{out}(\text{unbox } t); 1 \rangle \xrightarrow{\text{in}_{p_1} v_1} \langle t_1; p_1 \rangle \xrightarrow{\text{in}_{p_2} v_2} \langle t_2; p_2 \rangle \xrightarrow{\text{in}_{p_n} v_n} \langle t_n; p_n \rangle
\]

with \( \langle t_n; p_n \rangle \xrightarrow{\text{in}_{p_{n+1}} v_{n+1}} \langle t_{n+1}; p_{n+1} \rangle \) so that \( \langle t_{n+1}; p_{n+1} \rangle \in F_{\omega_1, \omega}(A, B) \). Since \( \xrightarrow{\text{f}} \) is deterministic, this defines the desired infinite sequence of reductions. Moreover, since \( \langle t_i; p_i \rangle \in F_{\omega_1, \omega}(A, B) \) for each \( i \), and the first index \( \mu \) decreases for every step of the form (a), we know that only finitely many reduction steps after \( \langle t_i; p_i \rangle \) 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 5.3 that \( \vdash \text{in}_{p_i} v_i : A + B \) for all \( i \geq 1 \).

5.3.4 Causality. The causality properties for the reactive step semantics are instances of the productivity, termination and liveness properties that we have proved above. Specifically, Lemmas 5.8, 5.9, and 5.10 are formulated in the context of an arbitrary input sequence \( \sigma \). For the proofs of the productivity, termination and liveness theorems, we instantiated \( \sigma \) with the sequence \( \sigma^{\omega} \) of empty inputs. The causality properties simply follow by instantiating \( \sigma \) with the sequence \( \sigma_{v_1}; \sigma_{v_2}; \sigma_{v_3}; \ldots \) constructed from the sequence of inputs \( v_1, v_2, v_3, \ldots \).

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.

Given \( v_1, v_2, v_3, \ldots \), let \( \sigma = \sigma_{v_2}; \sigma_{v_3}; \ldots \). By Corollary 5.6, \( \text{unbox } t \in T \llbracket \text{Str}(A) \rightarrow \text{Str}(B) \rrbracket_{\omega_1, v}^\sigma \) and \( t \sigma_{v_1} = t \) because \( t \) is well-typed and thus contains no input locations. Consequently, \( \text{unbox } t \sigma_{v_1} \in T \llbracket \text{Str}(A) \rightarrow \text{Str}(B) \rrbracket_{\omega_1, v}^\sigma \). Moreover, we can show that \( l^* \in T \llbracket \text{Str}(A) \rrbracket_{\omega_1, v + 1}^{\sigma_{v_1} \sigma} \), and therefore \( \text{adv } l^* \sigma_{v_1} \in T \llbracket \text{Str}(A) \rrbracket_{\omega_1, v}^{\sigma_{v_1} \sigma} \). Consequently, \( \text{unbox } t \text{ (adv } l^*) \in T_v^{\sigma_{v_1} \sigma} (B) \) for all \( v \). Similarly to the proof of Theorem 4.1, we can then use Lemma 5.8 to construct the infinite sequence of reduction steps

\[
\text{unbox } t \text{ (adv } l^*) \sigma_{v_1} \xrightarrow{\text{Str}} t_1 \xrightarrow{v_1'} t_1 \sigma_{v_2} \xrightarrow{\text{Str}} t_2 \xrightarrow{v_2'} t_2 \sigma_{v_3} \xrightarrow{\text{Str}} t_3 \xrightarrow{v_3'} \cdots
\]

which is just a different notation for the infinite sequence of reduction steps

\[
\text{unbox } t \text{ (adv } l^*) \xrightarrow{\text{Str}} t_1 \xrightarrow{v_1'} t_1 \xrightarrow{v_2'} t_2 \xrightarrow{v_3'} \cdots
\]

Moreover, if \( B \) is a value type, then \( \vdash v_i' : B \) follows from Lemma 5.3 for all \( i \geq 1 \).

6 RELATED WORK

The work by Cave et al. [2014] mentioned in the introduction defines a language with a modal operator \( \Box \) 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 \( \omega \). The relations corresponding to finite ordinals describe reductions up to finitely many steps, and the one at \( \omega \) describes global behaviour. They give an interpretation of types as predicates on values indexed by ordinals up to and including \( \omega \), and similarly to our interpretation of types, the interpretation of \( \Box A \) at \( \omega \) refers to the interpretation of \( A \) also at \( \omega \). Using this they prove strong normalisation, and sketch proofs of causality, productivity and liveness. 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 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 leaks, i.e., the problem of programs holding on to memory while continually allocating until they run out of space, and implicit time leaks, i.e., the problem of programs becoming gradually slower. 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 by evaluating programs on a machine with aggressive garbage collection. Bahr et al. [2019] recast this work in the setting a Simply RaTT, which unlike Krishnaswami [2013] uses Fitch style for programming with modal types, and extends 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 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 [Birkedal et al. 2017] 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. 2018; 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 $A \rightarrow \triangleright A$ 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 proving results on the lack of implicit space leaks in Lively RaTT. Although Lively RaTT is based on Simply RaTT, the corresponding results proved for this by Bahr et al. [2019] do not directly transfer because Lively RaTT allows multiple time ticks in a context. Still we have chosen to keep the restrictions known to be necessary for eliminating space leaks (e.g. disallowing unrestricted saving of data for later time steps) in Lively RaTT, with the hope of proving these properties in future work.

Finally, the \( \mathcal{U} \) types of Lively RaTT should be special cases of a class of temporal inductive types to be investigated in future work.

REFERENCES

Andreas Abel and Brigitte Pientka. 2013. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In Proceedings ICFP 2013. ACM, 185–196.

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 Vuillon. 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. 109–122.

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. ACM SIGPLAN Notices 48, 9 (2013), 197–208.

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.

Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2018. Guarded Cubical Type Theory. Journal of Automated Reasoning (Jun 2018).

Lars Birkedal, Hans Bugge Grathwohl, Ales Bizjak, and Ranald Clouston. 2017. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. Logical Methods in Computer Science 12 (2017).

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

Ales Bizjak, Lars Birkedal, and Marino Miculan. 2014. A model of countable nondeterminism in guarded type theory. In Rewriting and Typed Lambda Calculi. Springer, 108–123.

Ales Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E Møgelberg, and Lars Birkedal. 2016. Guarded dependent type theory with coinductive types. In International Conference on Foundations of Software Science and Computation Structures. Springer, 20–35.

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 International Conference on Foundations of Software Science and Computation Structures. Springer, 258–275.

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

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP ’97). ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258984.258973

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

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

, Vol. 1, No. 1, Article . Publication date: March 2020.
26 Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg

J. Hughes, L. Pareto, and A. Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In Conference Record of POPL ’96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. 410–423.

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.

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.

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/10.1145/2503656.2503665

Saunders MacLane and Ieke Moerdijk. 2012. Sheaves in geometry and logic: A first introduction to topos theory. Springer Science & Business Media.

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.

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bismulation as path type for guarded recursive types. Proceedings of the ACM on Programming Languages 3, POPL (2019), 1–29.

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 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). IEEE, 46–57.

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. 233–242.

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite step-indexing: Decoupling concrete and logical steps. In European Symposium on Programming. Springer, 727–751.

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

Niccolo Veltri and Andrea Vezzosi. 2020. Formalizing $\pi$-calculus in guarded cubical Agda. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. 270–283.