Abstract

Functional Reactive Programming (FRP) models reactive systems with events and signals, which have previously been observed to correspond to the “eventually” and “always” modalities of linear temporal logic (LTL). In this paper, we define a constructive variant of LTL with least fixed point and greatest fixed point operators in the spirit of the modal μ-calculus, and give it a proofs-as-programs interpretation in the realm of reactive programs. Previous work emphasized the propositions-as-types part of the correspondence between LTL and FRP; here we emphasize the proofs-as-programs part by employing structural proof theory. We show that this type system is expressive enough to enforce liveness properties such as the fairness of schedulers and the eventual delivery of results. We illustrate programming in this language using (co)iteration operators. We prove type preservation of our operational semantics, which guarantees that our programs are causal.

We give also a proof of strong normalization which provides justification that this type system is expressive enough to enforce liveness properties. A strong normalization proof using Tait’s method of saturated interpretations for the first n time steps, it allows one to evaluate programs one time step at a time. Moreover, we prove type preservation of our language. As a consequence, our language is causal; future inputs do not affect present results.

We prove type preservation of our language. As a consequence, our language is causal; future inputs do not affect present results.

1. Introduction

Reactive programming seeks to model systems which react and respond to input such as games, print and web servers, or user interfaces. Functional reactive programming (FRP) was introduced by Elliott and Hudak [9] to raise the level of abstraction for writing reactive programs, particularly emphasizing higher-order functions. Today FRP has several implementations [6–8, 27]. Many allow one to write unimplementable non-causal functions, where the present output depends on future input, and space leaks are all too common.

Recently there has been a lot of interest in type-theoretic foundations for (functional) reactive programming [13][16][20][22] with the intention of overcoming these shortcomings. In particular, Jeffrey [13] and Jeltsch [16] have recently observed that Pnueli’s linear temporal logic (LTL) [29] can act as a type system for FRP.

In this paper, we present a novel logical foundation for discrete time FRP with (co)iteration operators which exploits the full expressiveness afforded by the proof theory (i.e. the universal properties) for least and greatest fixed points, in the spirit of the modal μ-calculus [19]. The “always”, “eventually” and “until” modalities of LTL arise simply as special cases. We do this while still remaining relatively conservative over LTL.

Moreover, we demonstrate that distinguishing between and interleaving of least and greatest fixed points is key to statically guarantee liveness properties, i.e. something will eventually happen, by type checking. To illustrate the power and elegance of this idea, we describe the type of a fair scheduler – any program of this type is guaranteed to be fair, in the sense that each participant is guaranteed that his requests will eventually be served. Notably, this example requires the expressive power of interleaving least fixed points and greatest fixed points, a construction due to Park [28], and which is unique to our system.

Our approach of distinguishing between least and greatest fixed points and allowing for iteration and coiteration is in stark contrast to prior work in this area: Jeffrey’s work [14][15] for example only supports less expressive combinators instead of the primitive (co)recursion our system affords, and only particular instances of our recursive types. Krishnaswami et al. employ a more expressive notion of recursion, which entails unique fixed points [20][22]. This means that their type systems are actually less expressive, in the sense that they cannot guarantee liveness properties about their programs. Our technical contributions are as follows:

• A type system which, in addition to enforcing causality (as in previous systems [13][21]), also enables one to enforce liveness properties; fairness being a particularly intriguing example. Moreover, our type system forms a sound proof system for LTL. While previous work [13] emphasized the propositions-as-types component of the correspondence between LTL and FRP, the present work additionally emphasizes the proofs-as-programs part of the correspondence through the lens of structural proof theory. Our type system bears many similarities to Krishnaswami’s recent work [29]. The crucial difference lies in the treatment of recursive types and recursion. Our work distinguishes between least and greatest fixed points, while Krishnaswami’s work collapses them.

• A novel operational semantics which provides a reactive interpretation of our programs. One can evaluate the result of a program for the first n time steps, and in the next time step, resume evaluation for the (n + 1)st result. It allows one to evaluate programs one time step at a time. Moreover, we prove type preservation of our language. As a consequence, our language is causal: future inputs do not affect present results.

• A strong normalization proof using Tait’s method of saturated sets which justifies that our programs are productive total functions. It also demonstrates that our programs satisfy liveness properties derived from their types. Notably, our proof tackles the full generality of interleaving fixed points, and offers a novel treatment of monotonicity.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications – Applicative (functional) languages

Copyright notice will appear here once ‘preprint’ option is removed.
The paper is organized as follows: To illustrate the main idea, limitations and power of our foundation, we give several examples in Sec. 2. In particular, we elaborate the implementation of two fair schedulers where our foundation statically guarantees that each request will eventually be answered. We then introduce the syntax (Sec. 3) of our language which features (co)iteration operators and explicit delay operators together with typing rules (Sec. 4). In Sec. 5 we describe the operational semantics and prove type preservation. In Sec. 6 we outline the proof of strong normalization. In Sec. 7 we discuss consequences of strong normalization and type preservation, namely causality, liveness, and productivity. We conclude with related work.

2. Examples

To illustrate the power of our language, we first present several motivating examples using an informal ML or Haskell-like syntax. For better readability we use general recursion in our examples, motivating examples using an informal ML or Haskell-like syntax.

We use the "next" modality of LTL to describe data available in the next time step. On the term level, we use the corresponding induction form • and elimination form let • x = e in e’ where x is bound to the value of the expression e in the next time step.

2.1 The "always" modality

Our first example app produces a stream of elements of type B, given a stream fs of functions of type A → B and a stream xs of elements of type A by applying the nth function in fs to the nth value in xs. Such streams are thought of as values which vary in time.

Here we use the □ type of LTL to describe temporal streams, where the nth value is available at the nth time step. □ A can be defined in terms of the ○ modality as follows using a standard definition as a greatest fixed point (a coinductive datatype).

codata □ A = _ ::_ of A × □ A

□ A has one constructor :: which is declared as an infix operator and takes an A now and recursively a □ A in the next time step.

The functions hd and tl can then be defined, the only caveat being that the type of tl expresses that the result is only available in the next time step:

hd : □ A → A
tl : □ A → □ □ A

Finally, we can implement the app function as follows:

app : □ (A → B) → □ A → □ B
app fs xs = let • fs' = tl fs • x's' = tl xs
  in ((hd fs) (hd xs)) :: (• (app fs' x's'))

We use the □ elimination form, let • to bind the variables fs' and x's to values of the remaining streams in the next time step. Our typing rules will guarantee that fs' and x's are only usuable underneath a •, which we will explain further in the following examples.

Such a program is interpreted reactively as a process which, at each time step, receives a function A → B and a value A and produces a value B. More generally, given n functions A → B and n values A, it can compute n values B.

2.2 The "eventually" modality

A key feature of our foundation is the distinction between least fixed points and greatest fixed points. In other words, the distinction between data and codata. This allows us to make a distinction between events that may eventually occur and events that must eventually occur. This is a feature not present in the line of work by Krishnaswami and his collaborators, Benton and Hoffman [20–23] – they have unique fixed points corresponding most closely to our greatest fixed points.

To illustrate the benefits of having both, least and greatest fixed points, we present here the definition of LTL’s ○ operator (read: “eventually”) as a data type, corresponding to a type of events in reactive programming:

data ○ A = Now of A | Later of ○ ○ A

The function evapp below receives an event of type A and a time-varying stream of functions A → B. It produces an event of type B. Operationally, evapp waits for the A to arrive, applies the function available at that time, and fires the resulting B event immediately:

evapp : ○ A → □(A → B) → ○ B
evapp ea fs = case ea of
  | Now x ⇒ Now ((hd fs) x)
  | Later ea' ⇒ let • ea'' = ea'
  • fs' = tl fs
  in Later • (evapp ea'' fs')

This is not the only choice of implementation for this type. Such functions could opt to produce a B event before the A arrives, or even long after (if B is something concrete such as bool).

However, all functions with this type (in our system) have the property that given that an A is eventually provided, it must eventually produce a B, although perhaps not at the same time. This is our first example of a liveness property guaranteed by a type. This is in contrast to the "weak eventually" modality present in other work, which does not guarantee the production of an event.

It is interesting to note that this program (and all the other programs we write) can rightly be considered proofs of their corresponding statements in LTL.

2.3 Abstract server

Here we illustrate an abstract example of a server whose type guarantees responses to requests. This example is inspired by a corresponding example in Jeffrey’s [15] recent work.

We wish to write a server which responds to two kinds of requests: Get and Put with possible responses OK and Error. We represent these:

data Req = Get | Put
data Resp = OK | Error

At each time step, a server specifies how to behave in the future if it did not receive a request, and furthermore, if it receives a request, it specifies how to respond and also how to behave in the future. This is the informal explanation for the following server type, expressed as codata (here, we use coinductive record syntax):

codata Server = {noreq : □ Server,
  some : Req → Resp × □ Server}

Now we can write the server program which responds to Get with OK and Put with Error:

server : Server
server = { noreq = • server,
  some = λr. if isGet r
    then (OK, • server)
    else (Error, • server)}
Above, we say that if no request is made, we behave the same in the next time step. If some request is made, we check if it is a Get request and respond appropriately. In either case, in the next time step we continue to behave the same way. More generally, we could opt to behave differently in the next time step by e.g. passing along a counter or some memory of previous requests.

It is clear that this type guarantees that every request must immediately result in a response, which Jeffrey calls a liveness guarantee. In our setting, we reserve the term liveness guarantee for something which has the traditional flavor of “eventually something good happens”. That is, they are properties which cannot be falsified after any finite amount of time, because the event may still happen. The present property of immediately providing a response does not have this flavor: it can be falsified by a single request which does not immediately receive a response. In our setting, liveness properties arise strictly from uses of inductive types (i.e. data, or \(\mu\) types) combined with the temporal \(\bigcirc\) modality, which requires something to happen arbitrarily (but finitely!) far into the future.

### 2.4 Causality-violating (and other bad) programs

We wish to disallow programs such as the following, which has the effect of pulling data from the future into the present; it violates a causal interpretation of such programs. Moreover, its type is certainly not a theorem of LTL for arbitrary \(A\!\!:\!\!\)!

\[
\text{predictor} \, : \bigcirc (A + B) \rightarrow A + B
\]

\[
\text{predictor} \, x = \text{let} \, \bullet \, x' = x \text{ in } x' \quad -- \text{does not typecheck}
\]

does not typecheck

Our typing rules disallow this program roughly by restricting variables bound under a \(\bullet\) to only be usable under a \(\bullet\), in much the same way as Krishnaswami [20].

Similarly, the type \(\bigcirc(A + B)\) expresses that either an \(A\) or a \(B\) is available in the next time step, but it is not known yet which one it will be. Hence we disallow programs such as the following by disallowing case analysis on something only available in the future:

\[
\text{predictor} \, : \bigcirc (A + B) \rightarrow A + B
\]

\[
\text{predictor} \, x = \text{let} \, \bullet \, x' = x \text{ in } x' \quad \text{case } x' \text{ of } -- \text{does not typecheck}
\]

\[
| \text{inl } a \Rightarrow \text{inl } (\bullet \, a) \\
| \text{inr } a \Rightarrow \text{inr } (\bullet \, b)
\]

Such a program would tell us now whether we will receive an \(A\) or a \(B\) in the future. Again this violates causality. Due to this interpretation, there is no uniform inhabitant of this type, despite being a theorem of classical LTL. Similarly, \(\bigcirc \bot \rightarrow \bot\) is uninhabited in our system; one cannot get out of doing work today by citing the end of the world tomorrow.

Although it would be harmless from the perspective of causality, we disallow also the following:

\[
\text{import} \, : \, A \rightarrow \bigcirc A
\]

\[
\text{import} \, x = \bullet \, x \quad -- \text{does not typecheck}
\]

does not typecheck

This is disallowed on the grounds that it is not uniformly a theorem of LTL. Krishnaswami and Benton allow it in [21], but disallow it later in [22] to manage space usage. Syntactically, this is accomplished by removing from scope all variables not bound under a \(\bullet\) when moving under a \(\bullet\). However, some concrete instances of this type are inhabited, for example the following program which brings natural numbers into the future:

\[
\text{import} \, : \, \text{Nat} \rightarrow \bigcirc \text{Nat}
\]

\[
\text{import} \, \text{Zero} = \bullet \, \text{Zero}
\]

\[
\text{import} \, (\text{Succ } n) = \text{let} \, \bullet \, n' = \text{import } n \text{ in } (\bullet \, (\text{Succ } n'))
\]

Our language does not have Nakano’s guarded recursion [12]

\[
\text{fix} \, : \bigcirc (A \rightarrow A) \rightarrow A
\]

because it creates for us undesirable inhabitants of inductive types (inductive types collapse into coinductive types). For example, the following would be an undesirable inhabitant of \(\bigcirc \text{Nat} A\) in which the \(A\) is never delivered:

\[
\text{never} : \bigcirc A
\]

\[
\text{never} = \text{fix} \, (\lambda x. \text{ Later } x) \quad -- \text{disallowed}
\]

Finally, the following program which attempts to build a constant stream cannot be elaborated into the formal language. While it is guarded, we must remove all local variables from scope in the bodies of recursive definitions, so the occurrence of \(x\) is out of scope.

\[
\text{repeat} : \, A \rightarrow \Box A
\]

\[
\text{repeat } x = \text{xs}
\]

where \(\text{xs} = x :: \bullet \text{xs} \quad -- x \text{ out of scope!}
\]

This is intentional – the above type is not uniformly a theorem of LTL, so one should not expect it to be inhabited. As Krishnaswami et al. [22] illustrate, this is precisely the kind of program which leads to space leaks.

### 2.5 Fair scheduling

Here we define a type expressing the fair interleavings of two streams, and provide examples of fair schedulers employing this type – this is the central example illustrating the unique power of the presented system. This is enabled by our system’s ability to properly distinguish between and interleave least and greatest fixed points (i.e. data and codata). Other systems in the same vein typically collapse least fixed points into greatest fixed points or simply lack the expressiveness of recursive types.

First we require the standard “until” modality of LTL, written \(A \, U \, B\). This is a sequence of \(A\), terminated with a \(B\). In the setting of reactive programming, Jeltsch [17] calls programs of this type processes – they behave as time-varying signals which eventually terminate with a value.

\[
\text{data } A \, U \, B = \text{ Same of } A \times (\bigcirc (A \, U \, B))
\]

| Switch \, of \, B

Notably, since this is an inductive type, the \(B\) must eventually occur. The coinductive variant is “weak until”; the \(B\) might never happen, in which case the \(A\)s continue forever.

We remark that without temporal modalities, \(A \, U \, B\) is isomorphic to \((\text{List } A) \times B\), but because of the \(\bigcirc\), it matters when the \(B\) happens.

We define also a slightly stronger version of “until” which requires at least one \(A\), which we write \(A \, U' \, B\).

\[
\text{type } A \, U' \, B = A \times \bigcirc (A \, U' \, B)
\]

We characterize the type of fair interleavings of a stream of \(A\)s and \(B\)s as some number of \(A\)s until some number of \(B\)s (at least one), until an \(A\), and the process restarts. This is fair in the sense that it guarantees infinitely many \(A\)s and infinitely many \(B\)s. As a coinductive type:

\[
\text{codata } \text{Fair } A \, B = \text{ In } \text{ of } (A \, U' \, B \, U' (B \, U' (A \times \bigcirc (\text{Fair } A \, B))))
\]

This type corresponds to the Büchi automaton in Figure[1] With this type, we can write the type of a fair scheduler which takes a stream of \(A\)s and a stream of \(B\)s and selects from them fairly. Here is the simplest fair scheduler which simply alternates selecting an \(A\) and \(B\):

\[
\text{sched } : \, \square A \rightarrow \square B \rightarrow \text{Fair } A \, B
\]

\[
\text{sched }\, \text{as } bs =
\]

2013/7/19
The reader may notice that this scheduler drops the A’s at odd position and the B’s at even position. This could be overcome if one has import for the corresponding stream types, but at the cost of a space leak.

```
let • as' = tl as
• bs' = tl bs
in
  In (Switch (hd bs, • (Switch (hd as',
    let • as'' = tl as'
    • bs'' = tl bs'
    in • (sched as'' bs''))))
```

However, this “dropping” behaviour could be viewed positively: in a reactive setting, the source of the A requests could have the option to re-send the same request or even modify the previous request after observing that the scheduler decided to serve a B request instead.

Next we illustrate a more elaborate implementation of a fair scheduler which serves successively more A’s each time before serving a B. Again the type requires us to eventually serve a B.

```
as : a a a a a a a a a a a a
bs : b b b b b b b b b b b
```

We will need a special notion of “timed natural numbers” to implement the countdown. In the Succ case, the predecessor is only available in the next time step:

```
data TNat = Zero | Succ (□ TNat)
```

We can write a function which imports TNats:

```
import : TNat → □ TNat
```

The scheduler is implemented in Figure 2. It involves two mutually recursive functions: cnt is structurally decreasing on a TNat, counting down how many A’s to produce, while the recursive call to sch2’ is guarded by the necessary Switches to guarantee productivity, and increments the number of A’s served the next time. sch2 kicks off the process starting with zero A’s.

To understand why type TNat is necessary, we note cnt is structurally decreasing on its first argument. This requires the immediate subterm n’ to be available in the next step, not the current.

The important remark here is that these schedulers can be seen to be fair simply by virtue of typechecking and termination/productivity checking. More precisely, this is seen by elaborating the program to use the (co)iteration operators available in the formal language, which we illustrate in the next section.

---

**Figure 1.** Scheduler automaton.

---

**Figure 2.** Fair scheduler

---

**Figure 3.** LTL Syntax

---

### 3. Syntax

The formal syntax for our language (Figure 3) includes conventional types such as product types, sum types and function types. Additionally the system has the □A modality for values which will have the type A in the next step and least and greatest fixed points, μ and ν types. Our convention is to write the letters A, B, C for closed types, and F, G for types which may contain free type variables.

```
Types A, B, F, G ::= 1 | F × G | F + G | A → F | □ F | μX.F | νX.F | X
Terms M, N ::= X | () | (M, N) | fat M | snd M | int M | inv M | case M of int x ⇒ N | inv y ⇒ N' | λx.M | MN | △M | let • x = M in N | inj M | iterX.F (x.M) N | out M | coitX.F (x.M) N | map (△.F) η M
```

```
Contexts Θ,Γ ::= • | Γ, x: A
Kind Contexts Δ ::= • | Δ, X : ∗
Type substitutions ρ ::= • | ρ, F/X
Morphisms η ::= • | η, (x.M)/X
```

---

Our term language is mostly standard and we only discuss the terms related to □A modality and the fixed points, μ and ν. B describes a term M which is available in the next time step. let • x = M in N allows us to use the value of M which is available in the next time step in the body N. Our typing rules will guarantee that the variable only occurs under the □A modality and the fixed points, μ and ν types. Our language also includes iteration operator iterX.F (x.M) N and coiteration operator coitX.F (x.M) N. Intuitively, the first argument x.M corresponds to the inductive invariant while N specifies how many times to unroll the fixed point. The introduction form for μ types is inj M, rolling up a term M. The elimination form for ν types is out M, unrolling M.
We note that the $X,F$ annotations on $\text{iter}$ and $\text{coit}$ play a key role during runtime, since the operational semantics of $\text{map}$ are defined using the structure of $F$ (see Sec. 5). We do not annotate $\lambda$ abstractions with their type because we are primarily interested in the operational behaviour of our language, and not e.g. unique typing. The term map $(\Delta,F) \eta N$ witnesses that $F$ is a functor, and is explained in more detail in the next sections.

Our type language with least and greatest fixed points is expressive enough that we can define the always and eventual modality e.g.:

$\Box A \equiv \nu X, A \times X$

$\Diamond A \equiv \mu X, A + X$

The definition of $\Box A$ expresses that when we unfold a $\Box A$, we obtain a value of type $A$ (the head of the stream) and another stream in the next time step. The definition of $\Diamond A$ expresses that in each time step, we either have a value of type $A$ now or a postponed promise for a value of type $A$. The use of the least fixed point operator guarantees that the value is only postponed a finite number of times. Using a greatest fixed point would permit always postponing and never providing a value of type $A$.

We can also express the fact that a value of type $A$ occurs infinitely often by using both greatest and least fixed points. Traditionally one expresses this by combining the always and eventually modalities, i.e. $\Box \Diamond A$. However, there is another way to express this, namely:

$$\inf A \equiv \nu X, Y, (A \times X + Y)$$

In this definition, at each step we have the choice of making progress by providing an $A$ or postponing until later. The least fixed point implies that one can only postpone a finite number of times. The two definitions are logically equivalent, but have different constructive content – they are not isomorphic as types. Intuitively, $\Box \Diamond A$ provides, at each time step, a handle on an $A$ to be delivered at some point in the future. It can potentially deliver several values of type $A$ in the same time step. On the other hand, $\inf A$ can provide at most one value of type $A$ at each time step. This demonstrates that the inclusion of general $\mu$ and $\nu$ operators in the language (in lieu of a handful of modalities) offers more fine-grained distinctions constructively than it does classically. We show here also the encoding of $\text{Server}$, $\text{A U B}$, $\text{A U U B}$ and $\text{Fair A B}$ which we used in the examples in Sec. 2.

$\text{Server} \equiv \nu X, (\text{O X} \times (\text{Resp} \to \text{Resp} \times X))$

$\text{A U B} \equiv \mu X, (B + A \times X)$

$\text{A U U B} \equiv A \times (A \cup \text{B})$

$\text{Fair A B} \equiv \nu X, ((\text{A U B} \cup \text{A U U B} \times X))$

Finally, to illustrate the relationship between our formal language which features (co)iteration operators and the example programs which were written using general recursion, we show here the program $\text{app}$ in our foundation. Here we need to use the pair of $f$ and $x$ as the coinduction invariant:

$\text{app} : (\Box (A \to B) \to \Box A \to \Box B)$

$\text{app } f x \equiv \text{co} x, (A \times X) (x, \text{let } f s' = \text{tl } (f s) \text{ in } \text{let } x' = \text{tl } (\text{snd } x) \text{ in } ((\text{hd } (\text{fst } x)) \times (\text{hd } (\text{snd } x)), \bullet (f s', x')))$

$) (f s, x)$

4. Type System

We define well-formed types in Fig. 4. In particular, we note that free type variables cannot occur to the left of a $\rightarrow$. That is to say, we employ a strict positivity restriction, in contrast to Krishnaswami’s guardedness condition [20].

We give a type assignment system for our language in Fig. 5 where we distinguish between the context $\Theta$ which provides types for variables which will become available in the next time step (i.e. when going under a $\bullet$) and the context $\Gamma$ which provides types for the variables available at the current time. The main typing judgment, $\Theta; \Gamma \vdash M : A$ asserts that $M$ has type $A$ given the context $\Theta$ and $\Gamma$.

In general, our type system is similar to that of Krishnaswami and collaborators. While in their work, the validity of assumptions at a given time step is indicated either by annotating types with a time [21] or by using different judgments (i.e. now, later, stable) [20], we separate assumptions which are valid currently from the assumptions which are valid in the next time step via two different contexts. We suggest that keeping assumptions at most one step from the future models the practice of reactive programming better than
a general time-indexed type system. Much more importantly, our foundation differs in the treatment of recursive types and recursion.

Most rules are standard. When traversing $\lambda x.M$, the variable is added to the context $\Gamma$ describing the fact that $x$ is available in the current time step. Similarly, variables in each of the branches of the case-expression, are added to $\Gamma$.

The interesting rules are the ones for $\Box$ modality and fixed points $\mu$ and $\nu$. In the rule $I_{\Box}$, the introduction rule for $\Box$ with corresponding constructor $\bullet$, provides a term to be evaluated in the next time step. It is then permitted to use the variables promised in the next time step, so the assumptions in $\Theta$ move to the “available” position, while the assumptions in $\Gamma$ are no longer available. In fact, the variables in $\Theta$ remain inaccessible until going under a $\bullet$ – this is how causality is enforced. Our motivation for forgetting about current assumptions $\Gamma$ and not allowing them to persist is that we wish to obtain a type system which is sound for LTL. Allowing $\Gamma$ to persist would make the unsound $A \rightarrow \Box A$ easily derivable.

In the work of Krishnaswami et. al [22], forgetting about current assumptions plays a key role in managing space usage. In the work of Krishnaswami et. al [22], forgetting about current assumptions $\Gamma$ and not allowing them to persist is that we wish to obtain a type system which is sound for LTL. Allowing $\Gamma$ to persist would make the unsound $A \rightarrow \Box A$ easily derivable. Similarly, variables in each of the branches of the case-expression, are added to the context $\Theta$ and we continue to check the body $N$ allowing it to refer to $x$ under $\bullet$.

The elimination form for $\mu$ is an iteration operator $\text{iter}$. Examining an instantiation of this rule for $\text{Nat} \equiv \mu x.X.1 + X$ may help to clarify this rule:

\[ \vdash x : 1 + C \vdash M : C \quad \Theta; \Gamma \vdash N : \mu x.1 + X \quad \Theta; \Gamma \vdash \text{iter}_{X.1+X} (x. N) : C \]

The term $M$ contains the base case and the step case ($M$ will typically be a case analysis on $x$), while $N$ specifies how many times to iterate the step case of $M$ starting at the base case of $M$. For example, we illustrate here the doubling function on natural numbers written using this iteration operator. We write $\Theta$ for the term $\text{inj} (\text{inl}())$ and $\text{suc}$ for the term $\text{inj} (\text{inr} m)$.

\[
\begin{align*}
db &= \text{iter}_{X.1+X} (x. \text{case } x \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } u \mapsto \text{suc} (\text{suc} u)) \\
double &= \lambda x. \text{Nat. } db 
\end{align*}
\]

We note that dropping $\Theta$ and $\Gamma$ in the first premise is essential. Intuitively, the bodies of recursive definitions need to be used at arbitrary times, while $\Gamma$ and $\Theta$ are only available for the current and next time steps, respectively. This is easiest to see for the coit rule, where keeping $\Gamma$ allows a straightforward derivation of $A \rightarrow \Box A$ (repeat), which is unsound for LTL and easily produces space leaks. Similarly, keeping $\Gamma$ in the iter rule allows one to derive the unsound $A \rightarrow \Box B \rightarrow A \cup B$ which says that if $A$ holds now, and $B$ eventually holds, then $A$ in fact holds until $B$ holds.

\[
\begin{align*}
\text{unsound} &: A \rightarrow \Box B \rightarrow A \cup B \\
\text{unsound} &\equiv \lambda a. \lambda eb. \text{iter}_{X.1+X} (y, \text{case } y \text{ of } \text{inl } b \mapsto \text{inj} (\text{inl } b) \mid \text{inr } u \mapsto \text{inj} (\text{inr} (a, u)) - a \text{ out of scope!}) \end{align*}
\]

The typing rules for $\mu$ and $\nu$ come directly from the universal properties of initial algebras and terminal coalgebras. The reader may find it clarifying to compare these rules to the universal properties, depicted here:

\[
\begin{align*}
F(\mu F) \rightarrow F(C) &\quad \text{inj} \\
\mu F \rightarrow \text{iter } f &\quad f \\
C \rightarrow \nu F &\quad \text{coit } f \\
F(C) \rightarrow F(\nu F) &\quad \text{out}
\end{align*}
\]

We remark that the primitive recursion operator below can be derived from our (co)iteration operators in the standard way, so we do not lose expressiveness by only providing (co)iteration. One can similarly derive a primitive corecursion operator:

\[
\vdash x : [C \times \mu X.F / X] \vdash M : C \\
\Theta; \Gamma \vdash N : \mu x.X.F \\
\Theta; \Gamma \vdash \text{rec}_{X} (x. M) : N : C
\]

The term $\text{map} (\Delta, F) \eta M$ witnesses that $F$ is a functor. It has the effect of applying the transformations specified by $\eta$ at the positions in $M$ specified by $F$. It is a generic program defined on the structure of $F$. While this term is definable at the meta-level using the other terms in the language, we opt to include it in our syntax, because doing so significantly simplifies the operational semantics and proof of normalization. In this term, $\Delta$ binds the free variables of $F$. It is illustrative to consider the case where $F(Y) = \text{list } Y = \mu X.1 + X \times X$. If $y : A \vdash M : B$ and $N : \text{list } A$, then $\text{map} (\text{Y}, \text{list } Y ) ((y, M) / Y) : \text{list } B$. In this case, map implements the standard notion of map on lists!

We define two notions of substitution: substitution for a “current” variable, written $[N/x]M$, and substitution for a “next” variable, written $[N/x]^* M$. The key case in their definition is the case for $\bullet M$. For current substitution $[N/x]^* M$, in well-scoped terms, $x$ cannot occur in $M$, so we define:

\[
[N/x] (\bullet M) = \bullet M
\]

For next substitution $[N/x]^* (\bullet M)$, in the body of $M$, $x$ becomes a current variable, so we can defer to current substitution, defining:

\[
[N/x]^* (\bullet M) = \bullet ([N/x] M)
\]

These definitions are motivated by the desire to obtain tight bounds on how substitution interacts with the operational semantics without having to keep typing information to know that terms are well-scoped. These substitutions satisfy the following typing lemmas.

**Lemma 1.** Substitution Typing

1. If $\Theta; \Gamma; x : A \vdash M : B$ and $\Theta; \Gamma \vdash N : A$ then $\Theta; \Gamma \vdash [N/x] M : B$

2. If $\Theta; \Gamma; x : A \vdash M : B$ and $\vdash \Theta ; N : A$ then $\Theta; \Gamma \vdash [N/x]^* M : B$

5. Operational Semantics

Next, we define a small-step operational semantics for our language using evaluation contexts. Since we allow full reductions, a redex can be under a binder and in particular may occur under a $\Box$ modality. We define evaluation contexts in Fig. 6. We index an evaluation context $E_k$ by a depth $k$ which indicates how many $\Box$ modalities we traverse. Intuitively, the depth $k$ tells us how far we have stepped in time - or to put it differently, at our current time, we know that we have at most taken $k$ time steps and therefore terms under $k$ $\Box$ modalities are available to us now.

Single step reduction is defined on evaluation contexts at a time $\alpha < \omega + 1$ (i.e. either $\alpha \in \mathbb{N}$ or $\alpha = \omega$) and states that we can step $E_k[M]$ to $E_{k}[N]$ where $M$ is a redex occurring at depth $k$ where
\[ E_{k+1} := \bullet E_k \]
\[ E_k := E_N \mid M E_k \mid \text{fst} E_k \mid \text{snd} E_k \mid (E_k, M) \mid (M, E_k) \]
\[ \lambda x. E_k \mid \text{case } E_k \text{ of inl } x \mapsto N \mid \text{inr } y \mapsto N' \]
\[ \text{let } x = E_k \mid \text{let } x = M \text{ in } E_k \mid \text{inj } E_k \]
\[ \text{iter}_X F (x, M) E_k \mid \text{coit}_X F (x, M) E_k \]
\[ \text{out } E_k \mid \text{map } (\Delta, F) \eta E_k \]

**Figure 6. Evaluation contexts**

\[(\lambda x. M) N \rightarrow [N/x]M \]
\[\text{fst}(M, N) \rightarrow M \]
\[\text{snd}(M, N) \rightarrow N \]
\[\text{case } (\text{inl } M) \text{ of } x \mapsto N_1 \mid \text{inr } y \mapsto N_2 \rightarrow [M/x]N_1 \]
\[\text{case } (\text{inr } M) \text{ of } x \mapsto N_1 \mid \text{inr } y \mapsto N_2 \rightarrow [M/y]N_2 \]
\[\text{let } x = M \text{ in } N \rightarrow [M/x]N \]
\[\text{iter}_X F (x, M) \rightarrow [N/x]M \]
\[\text{map } (\Delta, F) (y, \text{coit}_X F (x, M) y) X \rightarrow [N/x]M \]

**Figure 7. Operational Semantics**

\(k \leq \alpha \text{ and } M \rightarrow N. \text{ More precisely, the reduction rule takes the following form:} \)
\[
\frac{M \rightarrow N}{E_k[M] \xrightarrow{\alpha} E_k[N]} \text{ if } k \leq \alpha
\]

If \(\alpha = 0\), then we are evaluating all redexes now and do not evaluate terms under a \(\bigcirc\) modality. If we advance to \(\alpha = 1\), we in addition need to contract all redexes at depth 1, i.e., terms occurring under one \(\bullet\), and so on. At \(\alpha = \omega\), we are contracting all redexes under any number of \(\bullet\). We have reached a normal form at time \(\alpha\) if for all \(k \leq \alpha\) all redexes at depth \(k\) have been reduced and no further reduction is possible.

The contraction rules for redexes (see Fig. 7) are mostly straightforward - the only exceptions are the iteration and coiteration rules. If we make an observation about a corecursive value by \((\text{coit}_X F (x, M) N)\), then we need to compute one observation of the resulting object using \(M\), and explain how to make more observations at the recursive positions specified by \(F\). Dually, if we unroll an iteration \((\text{iter}_X F (x, M) N)\), we need to continue performing the iteration at the recursive positions of \(N\) (the positions are specified by \(F\)), and reduce the result using \(M\).

Performing an operation at the positions specified by \(F\) is accomplished with \(\text{map}\), which witnesses that \(F\) is a functor. The operational semantics of \(\text{map}\) are presented in Fig. 8 They are driven by the type \(F\). Most cases are straightforward and more or less forced by the typing rules. The key cases, and the reason for putting \(\text{map}\) in the syntax in the first place, are those for \(\mu\) and \(\nu\). For \(\mu\), we reduce \(N\) until it is of the form \(\text{inj } N\). At which point, we can continue applying \(\eta\) inside \(N\), where now at the recursive positions specified by \(Y\) we need to continue recursively applying \(\text{map}\). The case for \(\nu\) is similar, except it is triggered when we demand an observation with out.

We remark that we do not perform reductions inside the bodies of \(\text{iter}\), \(\text{coit}\), and \(\text{map}\), as these are in some sense timeless terms (they will be used at multiple points in time), and it is not clear how our explicitly timed notion of operational semantics could interact with these. We sidestep the issue by disallowing reductions inside these bodies.

To illustrate the operational semantics and the use of map, we consider an example of a simple recursive program: doubling a natural number.

**Example** We revisit here the program \(\text{double}\) which multiplies a given natural number by two given in the previous section. Recall the following abbreviations for natural numbers: \(0 = \text{inj } (\text{inl }())\), \(\text{suc } w = \text{inj } (\text{inr } w), 1 = \text{suc } 0\), etc.

Let us first compute \(\text{double } 0\).

\[
\text{double } 0
\rightarrow \text{db } (\text{inj } (\text{inl }()))
\rightarrow \text{case } M_0 \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\]

where
\[
M_0 = \text{map } (1 + X) (y, \text{db } y/X) (\text{inl }())
\rightarrow^* \text{case } (\text{inl }()) \text{ of } \text{inl } v \mapsto \text{inl }() \mid \text{inr } u \mapsto \text{inr } (\text{db } u)
\rightarrow^* \text{case } (\text{inl }()) \text{ of } \text{inr } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\rightarrow 0
\]

We now compute \(\text{double } 1\).

\[
\text{double } 1
\rightarrow \text{db } (\text{inj } (\text{inr }0))
\rightarrow \text{case } M_1 \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\]

where
\[
M_1 = \text{map } (1 + X) (y, \text{db } y/X) (\text{inr }0)
\rightarrow^* \text{case } (\text{inl }0) \text{ of } \text{inl } v \mapsto \text{inl }() \mid \text{inr } u \mapsto \text{inr } (\text{db } u)
\rightarrow^* \text{case } (\text{inr }0) \text{ of } \text{inr } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\rightarrow \text{suc } (\text{suc }0) = 2
\]

We have the following type soundness result for our operational semantics:

**Theorem 2** (Type Preservation). For any \(\alpha\), if \(M \xrightarrow{\alpha} N\) and \(\Theta; \Gamma \vdash M : A\) then \(\Theta; \Gamma \vdash N : A\).

Observe that after evaluating \(M \xrightarrow{\alpha} N\), where \(N\) is in normal form, one can then resume evaluation \(N \xrightarrow{\alpha+1} N'\) to obtain the cumulative result available at the next time step. One may view this as restarting the computation, aiming to compute the result up to \(n+1\), but with the results up to time \(n\) memoized. In practical implementations, one is typically only concerned with \(\sim_0\) (see below), however considering the general \(\sim_\alpha\) gives us the tools to analyze programs from a more global viewpoint, which is important for liveness guarantees.

Our definition of substitution is arranged so that we can prove the following bounds on how substitution interacts with \(\sim_\alpha\), which are important in our proof of strong normalization. Notice that these are independent of any typing assumptions.

**Proposition 3.**
1. If \(N \sim_\alpha N'\) then \([N/x]\ M \sim_\alpha [N'/x]\ M\)
2. If \(M \sim_\alpha M'\) then \([N/x]\ M \sim_\alpha [N/x]\ M'\)
3. If \(N \sim_\alpha N'\) then \([N/x]\ M \sim_\alpha [N'/x]\ M\)
4. If \(N \sim_\alpha N'\) then \([N/x]\ M \sim_\alpha [N'/x]\ M\)
5. If \(M \sim_\alpha M'\) then \([N/x]\ M \sim_\alpha [N/x]\ M'\)

Our central result is a proof of strong normalization for our calculus which we prove in the next section.

### 6. Strong Normalization

In this section we give a proof of strong normalization for our calculus using the Girard-Tait reducibility method [10] [11] [31].
In our setting, this means we prove that for any $\alpha \leq \omega$, every reduction sequence $M \leadsto_{\alpha} \ldots$ is finite, for well-typed terms $M$. This means that one can compute the approximate value of $M$ up to time $\alpha$. In fact, this result actually gives us more: our programs are suitably causal using normalization at $\omega$, and the type of a program gives rise to a liveness property which it satisfies, using normalization at $\omega$. Our logical relation is in fact a degenerate form of Kripke logical relation, as we index it by an analysis of redexes separate from the more semantic parts of the sequences rooted at $M$. This means we prove that for any condition on $\alpha$ when $\omega$ normalizes at program gives rise to a liveness property which it satisfies, using normalization at $\omega$. In fact, this result actually gives us more: our programs are suitably causal using normalization at $\omega$. This means that one can compute the approximate value of $M$ for all $\alpha$. One can easily verify by induction that if $\eta(X) = x.M$ then $\eta \in \Delta.X$.

**Definition 5.** We define the following next step operator on indexed sets $A : \omega + 1 \to \mathcal{P}(tm)$:

- $\bullet A^0 \equiv tm$
- $\bullet A^m+1 \equiv A^m$
- $\bullet A^\omega \equiv A^\omega$

The motivation for this is that it explains what happens when we go under a $\bullet$ -- we are now interested in reducibility under one fewer $\bullet$. Below, we define a notion of normalizing weak head reduction. We write $M \leadsto_{\alpha} M'$ for a contraction, and $M \leadsto_{\alpha} N$ for a contraction occurring under a weak head context. This is a weak head reduction where every term which may be lost along the way (e.g. by a vacuous substitution) is required to be strongly normalizing. This is a characteristic ingredient of the saturated set method. It is designed this way so as to (backward) preserve strong normalization. The intuition is that weak head redexes are unavoidable -- reducing other redexes can only postpone a weak head reduction, not eliminate it. We define below weak head reduction contexts and normalizing weak head reduction. We show only the lambda case. The full definition can be found in the long version of the paper.

$$\mathcal{H} ::= \emptyset \mid \text{fst } \mathcal{H} \mid \text{snd } \mathcal{H} \mid \mathcal{H} N \mid \text{iter } X : F (x.M) \mathcal{H} \mid \text{map } (\Delta. \mu Y.F) \eta \mathcal{H} \mid \text{out } (\mathcal{H} M) \mathcal{H} \mid \lambda x.M \mathcal{H} \mid \eta y$$

**Lemma 6.** $sn_{\alpha}$ is backward closed under normalizing weak head reduction: If $M' \in sn_{\alpha}$ and $M \leadsto_{\alpha} M'$ then $M \in sn_{\alpha}$.

We note that in all of our proofs, the subscript $\alpha$ plays little to no role, except of course in the cases pertaining to the next step operator $\bullet$. For this reason, we typically highlight the $\bigcirc$ cases of the proofs.

**Definition 7.** We define $sn_{\alpha} = \{ \mathcal{H}[x] \in sn_{\alpha} \}$

We can now define saturated sets as subsets of strongly normalizing terms. The rest of the proof proceeds by showing that our types can be interpreted as saturated sets, well-typed terms inhabit saturated sets, and hence are strongly normalizing.

**Definition 8.** Saturated sets. An indexed set $A : \omega + 1 \to \mathcal{P}(tm)$ is saturated if

1. $A \subseteq sn$
2. For any α, M, M′, if $M \leadsto^*_\alpha M'$ and $M' \in A^\alpha$ then $M \in A^\alpha$ (Backward closure under normalizing weak head reduction)

3. $\text{sne} \subseteq A$

It is immediate from Lemma 9 that $\text{sne}$ is saturated.

**Definition 9.** For an indexed set $A_\omega : \omega + 1 \rightarrow \mathcal{P}(tm)$, we define $\overline{A}$ as its closure under conditions 2 and 3.

$$\overline{A} = \{M|\exists M'. M \leadsto^*_\alpha M' \land (M' \in A^\alpha \lor M' \in \text{sne}_\alpha)\}$$

To interpret least and greatest fixed points, we construct a complete lattice structure on saturated sets:

**Lemma 10.** Saturated sets form a complete lattice under $\subseteq$, with greatest lower bounds and least upper bounds given by:

$$\bigwedge S \equiv (\bigcap S) \cap \text{sne} \quad \bigvee S \equiv \bigcup S$$

For $\wedge$, we intersect with $\text{sne}$ so that the nullary lower bound is $\text{sne}$, and saturated. For non-empty $S$, we have $\bigwedge S = \bigcap S$. As a consequence, by an instance of the Knaster-Tarski fixed point theorem, we have the following theorem:

**Corollary 11.** Given $F$ which takes predicates to predicates, we define:

$$\mu F \equiv \bigwedge \{C \text{ saturated} \mid F(C) \subseteq C\}$$

$$\nu F \equiv \bigvee \{C \text{ saturated} \mid C \subseteq F(C)\}$$

If $F$ is monotone and takes saturated sets to saturated sets, then $\mu F$ (resp. $\nu F$) is a least (resp. greatest) fixed point of $F$ in the lattice of saturated sets

The same operator definitions are convenient, as they allow us to reason at a high level of abstraction without having to introduce $\alpha$ at several places in the proof.

**Definition 12.** We define:

$$(A \circ f)^\alpha \equiv \{M|f M \in A^\alpha\}$$

$$(A \star f)^\alpha \equiv \{fM|M \in A^\alpha\}$$

We will often use these notations for partially applied syntactic forms, e.g. $\text{sne} \star \text{inj}$

We are now in a position to define the operators on saturated sets which correspond to the operators in our type language.

**Definition 13.** We define the following operations on saturated sets:

1. $A \times B \equiv \text{sne}$

2. $(A \circ f)\alpha \equiv (A \circ f) \cap (B \circ \text{snd})$

3. $(A + B)\alpha \equiv (A \star \text{inj} ) \cup (B \star \text{inr} )$

4. $(A \rightarrow B)^\alpha \equiv \{M|\forall N \in A^\alpha, (MN) \in B^\alpha\}$

5. $(\bigcirc A)^\alpha \equiv (\bigcirc A) \star \bullet$

6. $\mu F \equiv \mu(X \mapsto F(X) \star \text{inj})$

7. $\nu F \equiv \nu(X \mapsto F(X) \circ \text{out})$

Notice that $\mu F$ is defined regardless of whether $F$ is monotone, although we only know that $\mu F$ is actually a least fixed point when $F$ is monotone. We remark also that our $\rightarrow$ definition does not resemble the Kripke-style definition one might expect. That is to say, we are using the discrete partial order on $\omega + 1$. We do not need the monotonicity that the standard ordering would grant us, since our type system does not in general allow carrying data into the future.

**Lemma 14.** The operators defined in Definition 13 take saturated sets to saturated sets.

We are now ready to interpret well-formed types as saturated sets. The definition is unsurprising, given the operators defined previously.

**Definition 15.** Given $\rho$, an environment mapping the free variables of $F$ to saturated sets, we define the interpretation $[F](\rho)$ of an open type as a saturated set as follows:

$$[X](\rho) \equiv \rho(X)$$

$$[1](\rho) \equiv 1$$

$$[F \times G](\rho) \equiv [F](\rho) \times [G](\rho)$$

$$[F + G](\rho) \equiv [F](\rho) + [G](\rho)$$

$$[A \rightarrow F](\rho) \equiv [A](\rho) \rightarrow [F](\rho)$$

$$[\bigcirc F](\rho) \equiv [\bigcirc F](\rho)$$

$$\mu X.F(\rho) \equiv \mu(X \rightarrow [F](\rho, X/X))$$

$$\nu X.F(\rho) \equiv \nu(X \rightarrow [F](\rho, X/X))$$

Observe that, by Lemma 14, every $[F](\rho)$ is saturated.

If $\theta = N_1/y_1, \ldots, N_n/y_n$, and $\sigma = M_1/x_1, \ldots, M_m/x_m$, we use $\sigma \Gamma M$ to mean that $M_i$ is saturated.

The following holds, where we assume $\Theta$, $\Gamma$, $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are saturated.

1. If $\Theta \equiv \mathcal{A}$ then $\Theta \equiv \mathcal{B}$.

2. If $\Theta; \Gamma \equiv \mathcal{B}$ then $\Theta; \Gamma \equiv \mathcal{A}$.

3. If $\mathcal{F}$ is a monotone function from saturated sets to saturated sets, $\alpha_i \Gamma M \equiv F(\mu \mathcal{F})$ then $\Theta; \Gamma \equiv \mathcal{F} M \equiv \nu \mathcal{F}$

4. If $\mathcal{F}$ is a monotone function from saturated sets to saturated sets, then $\Theta; \Gamma \equiv M \equiv \nu \mathcal{F}$.

**Proof.** We show only the case for $\nu$. The rest are straightforward or standard.

1. We are given $\Theta \equiv \mathcal{A}$, i.e., for any $\alpha$ and $\theta \in \Theta^\alpha$, that $[\bullet; \theta]M \in A^\alpha$.

   Suppose we are given $\alpha$ and $\theta \in (\bullet \Theta)^\alpha$ and $\sigma \in \Theta^\alpha$.

   Case 0: Then $(\bullet \Theta)^\alpha = \text{tm}$, so $[\bullet; \theta]M \in (\bullet \Theta)^\alpha$.

   Case $\alpha = m + 1$: Then $(\bullet \Theta)^{m+1} = A^m$, and $\theta \in \Theta^m$. By our assumption, taking $\alpha = m$, we have $[\bullet; \theta]M \in A^m$.

   Case $\alpha = \omega$: Then $(\bullet \Theta)^\omega = A^\omega$, and $\theta \in \Theta^\omega$. By our assumption, $[\bullet; \theta]M \in A^\omega$.

   In any case, we have $[\bullet; \theta]M \in (\bullet \Theta)^\alpha$.

   Then $[\theta; \sigma](\bullet \Theta)M = \mu \left([\bullet; \theta]M \equiv (\bullet \Theta) \sigma\right)$

   Hence $[\theta; \sigma](\bullet \Theta)M \equiv (\bullet \Theta)^\alpha$ as required.

With this lemma, it remains only to handle $\text{map}$, $\text{iter}$, and $\text{coit}$. Below we show the semantic typing lemma for map. We write $\rho_1 \equiv \eta : \rho_2$ where $\eta = (x.M_1/X_1, \ldots, x.M_m/X_m)$ and $\rho_1 = A_1/X_1, \ldots, A_m/X_m$ and $\rho_2 = B_1/X_1, \ldots, B_n/X_n$ and the $A_i$ and $B_i$ are saturated to mean $x_i : A_i = B_i$ for each $i$. Notably, because we define map independently of $\text{iter}$ and $\text{coit}$, we can prove this directly before tackling $\text{iter}$ and $\text{coit}$, offering a simplification.
of known proofs for interleaving $\mu \nu$. For readability, we write $F$ instead of $(A, F)$.

**Lemma 18** (Semantic typing for map). If $\rho_1 \vdash \eta : \rho_2$ then $x : [F](\rho_1) \vdash\text{map } F \eta : [F](\rho_2)$

Proof. Notice by unrolling the definitions, this is equivalent to showing $[F](\rho_1) \subseteq [F](\rho_2) \circ (\text{map } F \eta)$. The proof proceeds by induction on $F$. We show only the case for $\mu$. The case for $\nu$ is analogous. The rest are straightforward using Lemma 17 and backward closure. We present the proof in more detail in the long version.

Case $\mu X : F$: This proceeds primarily using the least fixed point property and $\rightarrow$ closure.

Let $C = \mu X . F(\rho_2) = \mu (X \mapsto [F](\rho_2, X^*) \circ \text{inj})$

Let $D = C \circ (\text{map } (\mu X . F) \eta)$

Let $N = (\text{map } F(\eta, \text{map } (\mu X . F) \eta))$

1. $D$ is saturated

2. $x : D \vdash (\mu X . F) \eta : C$ (by definition of $\vdash$, $\beta$)

3. $x : [F](\rho_1, D) \vdash F(\eta, \text{map } (\mu X . F) \eta) : [F](\rho_2, C)$ (by IH)

4. $[F](\rho_1, D) \subseteq [F](\rho_2, C) \circ (\text{map } F(\eta, \text{map } (\mu X . F) \eta))$ (by definition of $\vdash$)

   $\subseteq [F](\rho_2, C) \circ \text{inj} \circ \text{inj} \circ N$

   $\subseteq [F](\rho_2, C) \circ \text{inj} \circ \text{inj} \circ N$ (by mon. of $\circ$)

   $= C \circ \text{inj} \circ N$ (by rolling fixed point)

   $= C \circ (\text{inj}(\text{map } F(\eta, \text{map } (\mu X . F) \eta) -))$ (by def)

   $\subseteq C \circ (\text{map } (\mu X . F) \eta \circ \text{inj})$

   $= D \circ \text{inj}$ (by def)

5. $[F](\rho_1, D) \circ \text{inj} \subseteq D$ (by adjunction $\ast$)

6. $[F](\rho_1, D) \circ \text{inj} \subseteq D$ (by adjunction $\lnot$)

7. $\mu X . F(\rho_1) \subseteq D$ (by lfp property)

8. $y : [\mu X . F(\rho_1)] \vdash (\mu X . F) \eta : [\mu X . F(\rho_2)]$ (by definitions of $\vdash$, $\beta$, $\alpha$)

The only term constructors remaining to consider are $\text{iter}$ and $\text{coit}$. These are the subject of the next two lemmas, which are proven similarly to the $\mu$ and $\nu$ cases of the map lemma. Namely, they proceed primarily by using the least (greatest) fixed point properties and backward closure under $\rightarrow$, appealing to the map lemma. The proofs can be found in the long version of the paper.

**Lemma 19.** If $x : [F](C / X) \vdash M : C$ where $C$ is saturated, we have $y : [\mu X . F] \vdash \text{iter}_{X,F} (\text{inj} \langle x, M \rangle) : y : C$

**Lemma 20.** If $x : C \vdash M : [F](C / X)$ where $C$ is saturated, we have $y : C \vdash \text{coit}_{X,F} (\text{inj} \langle x, M \rangle) : y : [\mu X . F]$.

By now we have shown that all of the term constructors can be interpreted, and hence the fundamental theorem is simply an induction on the typing derivation, appealing to the previous lemmas.

**Theorem 21** (Fundamental theorem). If $\Theta; \Gamma \vdash M : A$, then we have $[\Theta; \Gamma] ; [F] \vdash M : [A]$.

**Corollary 22.** If $\Theta; \Gamma \vdash M : A$ then for any $\alpha$, $M$ is strongly normalizing at $\alpha$.

7. **Causality, Productivity and Liveness**

We discuss here some of the consequences of type soundness and strong normalization and explain how our operational semantics enables one to execute programs reactively.

We call a term $M$ a $\alpha$-value if it cannot step further at time $\alpha$. We may write this $M \not\rightarrow_{\alpha}$. We obtain as a consequence of strong normalization that there is no closed inhabitant of the type $\perp$ (which we define as $\mu X . X$), since there is no closed $\alpha$-value of this type. Perhaps surprisingly, we can also show there is no closed inhabitant of $\perp \perp \rightarrow \perp$. For if there was some $\nu : F : \perp \perp \rightarrow \perp$, we could evaluate $\nu : \perp \perp \vdash \text{map } (\perp \perp) \cdot \perp \perp$ at $0$ to obtain a $0$-value $\perp \perp \vdash v : \perp \perp$, which by inspection of the possible $0$-values, cannot exist!

Similarly, we can demonstrate an interesting property of the type $B \equiv \mu X . C \otimes X$. First, there is no closed term of type $B \rightarrow \perp$. For if there was some $f : B \rightarrow \perp$, we could normalize $x : B ; \vdash f(\text{inj } x) : B$ at time $0$ to obtain a $0$-value $x : B \vdash v : \perp \perp$, which cannot exist. Moreover, there is no closed term of type $B$, since there is no closed $\omega$-value of type $B$. That is, $B$ is neither inhabited nor provably uninhabited inside the logic.

To show how our operational semantics and strong normalization give rise to a causal (reactive) interpretation of programs, as well as an explanation of the liveness properties guaranteed by the types, we illustrate here the reactive execution of an example program $x_0 : \otimes P \vdash M_0 : \otimes Q$. Such a program can be thought of as waiting for a $P$ event from its environment, and at some point delivering a $Q$ event. For simplicity, we assume that $P$ and $Q$ are pure (non-temporal) types such as $\text{Bool}$ or $\mathbb{N}$. We consider sequences of interaction which begin as follows, where we write $\text{now } p$ for $\text{inj } (\text{inj } p)$ and later $\bullet t$ for $\text{inj } (\text{inj } t)$.

$$\begin{align*}
\text{[later } \bullet x_1/x_0] M_0 & \rightsquigarrow_* \text{[later } \bullet M_1 \\
\text{[later } \bullet x_2/x_1] M_1 & \rightsquigarrow_* \text{[later } \bullet M_2 \\
\vdots & \\
\end{align*}$$

Each such step of an interaction corresponds to the environment telling the program that it does not yet have the $P$ event in that time step, and the program responding saying that it has not yet produced a $Q$ event. Essentially, at each stage, we leave a hole $x_i$ standing for input not yet known at this stage, which we will reframe further in the next time step. The resulting $M_i$ acts as a continuation, specifying what to compute in the next time step. We note that each $x_i : \otimes P \vdash M_i : \otimes Q$ by type preservation. Such a sequence may not end, if the environment defers providing a $P$ forever. However, it may end one of two ways. The first is if eventually the environment supplies a closed value $p$ of type $P$:

$$\text{now } p \vdash M_{i+1} \rightsquigarrow_* v \not\rightarrow_{\omega}$$

In this case, $\vdash \text{now } p \vdash M_{i+1} \rightsquigarrow Q$. By type preservation and strong normalization, we can evaluate this completely to $\vdash v : \otimes Q$. By an inspection of the closed $\omega$-values, $v$ must be of the form later(\text{later} \bullet (\text{later} \bullet (\text{now } q)))$. That is, a $Q$ is eventually delivered in this case.

The second way such an interaction sequence may end is if the program produces a result before the environment has supplied a $P$:

$$\text{[later } \bullet x_{i+1}/x_i] M_i \rightsquigarrow_* \text{now } q$$

We remark that type preservation of $\not\rightarrow_{\omega}$ provides an explanation of causality: since $x_{i+1} : \otimes P_1 ; \vdash \text{[later } \bullet x_{i+1}/x_i] M_i : \otimes Q$, if evaluating the term $\text{[later } \bullet x_{i+1}/x_i] M_i$ with $\not\rightarrow_{\omega}$ produces a $0$-value $v$, then $x_{i+1} : \otimes P_1 ; \vdash v : \otimes Q$ and by an inspection of the $0$-values of this type, we see that $v$ must be of the form later(\text{later} \bullet M_{i+1}) or now $p$ since the variable $x_{i+1}$ is in the next context, it cannot interfere with the part of the value in the present, which means the present component cannot be stuck on $x_{i+1}$. That is, $x_{i+1}$ could only possibly occur in $M_{i+1}$. This illustrates that future inputs do not affect present output – this is precisely what we mean by causality.

We also remark that strong normalization of $\not\rightarrow_{\omega}$ guarantees reactive productivity. That is, the evaluation of $\text{[later } \bullet x_{i+1}/x_i] M_i$ is
guaranteed to terminate at some 0-value $v$ by strong normalization. As an aside, we note that if we were to use a time-indexed type system such as that of Krishnaswami and Benton [21], one could generalize this kind of argument to reduction at $n$, and normalize terms using $\eta$, to obtain $n$-value where $x$ can only occur at time step $n+1$. This gives a more global perspective of reactive execution. However, we use our form of the type system because we find it corresponds better in practice to how one thinks about reactive programs (one step at a time).

Finally, strong normalization of $\eta$ provides an explanation of liveness. When evaluating a closed term $M : Q \omega$ at $\omega$, we arrive at a closed $\omega$-value $v : Q$. By inspection of the normal forms, such a value must provide a result after only finitely many delays. This is to say that when running programs reactively, the environment may choose not to satisfy the requisite liveness requirement (e.g. it may never supply a $P$ event). In which case, the output of the program cannot reasonably be expected to guarantee its liveness property, since it may be waiting for an input event which never comes. However, we have the conditional result that if the environment satisfies its liveness requirement (e.g. eventually it delivers a $P$ event) then the result of the interaction satisfies its liveness property (e.g. eventually the program will fire a $Q$ event).

8. Related Work

Most closely related to our work is the line of work by Krishnaswami and his collaborators [20, 22]. Our type systems are similar; in particular our treatment of the $\bigcirc$ modality and causality. The key distinction lies in the treatment of recursion and fixed points. Krishnaswami et al. employ a Nakano-style guarded recursion rule which allows some productive programs to be written more conveniently than with our (co)iteration operators. However, their recursion rule has the effect of collapsing least fixed points into greatest fixed points. Both type systems can be seen as proofs-as-programs interpretations of temporal logics; ours has the advantage that it is capable of expressing liveness guarantees (and hence it retains a tighter relationship to LTL). In their recent work [20, 22], they obtain also promising results about space usage, which we have so far ignored in our foundation. In his most recent work, Krishnaswami [20] describes an operational semantics with better sharing behaviour than ours.

Another key distinction between the two lines of work is that where we restrict to fixed points of strictly positive functors, Krishnaswami restricts to guarded fixed points – type variables always occur under $\bigcirc$; even negative occurrences are permitted. In contrast, our approach allows a unified treatment of typical pure recursive datatypes (e.g. list) and temporal types (e.g. $\square A$), as well as more exotic “mixed” types such as $\bigcirc X, X + \bigcirc X$. Krishnaswami observes that allowing negative, guarded occurrences enables the definition of guarded recursion. As a consequence, this triggers a collapse of (guarded) $\mu$ and $\nu$, so negative occurrences appear incompatible with our goals.

Also related is the work of Jeffrey [14, 15] and Jeltsch [16], who first observed that LTL propositions correspond to FRP types. Both consider also continuous time, while we focus on discrete time. Here we provide a missing piece of this correspondence: the proofs-as-programs component. Jeffrey writes programs with a set of causal combinators, in contrast to our ML-like language. He systems lack general (co)recursive datatypes and as a result, one cannot write practical programs which enforce some liveness properties such as fairness as we do here. In his recent work [15], he illustrates what he calls a liveness guarantee. The key distinction is that our liveness guarantees talk about some point in the future, while Jeffrey only illustrates guarantees about specific points in time. We illustrated in Sec. 4 that our type system can also provide similar guarantees. We claim that our notion of liveness retains a tighter correspondence to the concept of liveness as it is defined in the temporal logic world.

Jeltsch [16, 17] studies denotational settings for reactive programming, some of which are capable of expressing liveness guarantees. He obtains the result that his Concrete Process Categories (CPCs) can express liveness when the underlying time domain has a greatest element. He does not describe a language for programming in CPCs, and hence it is not clear how to write programs which enforce liveness properties in CPCs, unlike our work. He discusses also least and greatest fixed points, but does not mention the interleaving case in generality, nor does he treat their existence or meta-theory as we do here. His CPCs may provide a promising denotational semantics for our calculus.

The classical linear $\mu$-calculus (often called $\nuTL$) forms the inspiration for our calculus. The study of (classical) $\nuTL$ proof systems goes back at least to Lichtenstein [23]. Our work offers a constructive, type-theoretic view on $\nuTL$.

Synchronous dataflow languages, such as Esterel [3], Lustre [5], and Lucid Synchrone [28] are also related. Our work contributes to a logical understanding of synchronous dataflow, and in particular our work could possibly be seen as providing a liveness-aware type system for such languages.

9. Conclusion

We have presented a type-theoretic foundation for reactive programming inspired by a constructive interpretation of linear temporal logic, extended with least and greatest fixed point type constructors, in the spirit of the modal $\mu$-calculus. The distinction of least and greatest fixed points allows us to distinguish between events which may eventually occur or must eventually occur. Our type system acts as a sound proof system for LTL, and hence expands on the Curry-Howard interpretation of LTL. We prove also strong normalisation, which, together with type soundness, guarantees causality, productivity, and liveness of our programs.

10. Future Work

Our system provides a foundational basis for exploring expressive temporal logics as type systems for FRP. From a practical standpoint, however, our system is lacking a few key features. For example, the import functions we wrote in Section 5 unnecessarily traverse the entire term. From a foundational perspective, it is interesting to notice that they can be implemented at all. However, in practice, one would like to employ a device such as Krishnaswami’s stability [20] to allow constant-time import for such types. We believe that our system can provide a solid foundation for exploring such features in the presence of liveness guarantees.

It would also be useful to explore more general forms of recursion with syntactic or typing restrictions such as sized types to guarantee productivity and termination/liveness, which would allow our examples to typecheck as they are instead of by manual elaboration into (co)iteration. Tackling this in the presence of interleaving $\mu$ and $\nu$ is challenging. This is one motivation for using Nakano-style guarded recursion [12]. A promising direction is to explore the introduction of Nakano-style guarded recursion at so-called complete types in our type system (by analogy with the complete ultrametric spaces of [21]) - roughly, types built with only $\nu$, not $\mu$. This would be a step in the direction of unifying the two approaches.

We are in the process of developing a denotational semantics for this language in the category of presheaves on $\omega + 1$, inspired by the work of Birkedal et al. [4] who study the presheaves on $\omega$, as well as Jeltsch [17] who studies more general (e.g. continuous) time domains. The idea is that the extra point “at infinity” expresses the global behaviour – it expresses our liveness guarantees, which can only be observed from a global perspective on time. The chief chal-
lengen in our setting lies in constructing interleaving fixed points. It is expected that such a denotational semantics will provide a crisper explanation of our liveness guarantees.

References