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 mu-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 the language is productive and that our programs satisfy liveness properties derived from their types.

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

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 [13] to raise the level of abstraction for writing reactive programs, particularly emphasizing higher-order functions. Today FRP has several implementations [10–12, 31]. 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 [12, 24–26] with the intention of overcoming these shortcomings. In particular, Jeffrey [17] and Jeltsch [20] have recently observed that Pnueli’s linear temporal logic (LTL) [34] 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 [23]. 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 [33], 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 [18, 19] 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 [24–26]. 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 [17, 25]), 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 [17] emphasized the propositions-as-types component of the correspondence between LTL and FRP, the present work additionally emphasizes the proof-as-programs part of the correspondence through the lens of structural proof theory. Our type system bears many similarities to Krishnaswami’s recent work [24]. 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.

Andrew Cave  Francisco Ferreira  Prakash Panangaden  Brigitte Pientka
McGill University
{acave1, fferre8, prakash, bpientka}@cs.mcgill.ca

[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. 2) of our language which features (co)iteration operators and explicit delay operators together with typing rules (Sec. 3). 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, although our foundation only provides (co)iteration operators. However, all the examples can be translated into our foundation straightforwardly and we subsequently illustrate the elaboration in Sec. 3.

On the type level, we employ a type constructor corresponding to the “next” modality of LTL to describe data available in the next time step. On the term level, we use the corresponding introduction form $\bullet$ and elimination form $\mathit{let} \ x = e \ in \ e'$ where $x$ is bound to the value of the expression $e$ in the next time step.

2.1 The “always” modality

Our first example $\mathit{app}$ produces a stream of elements of type $B$, given a stream $fs$ of functions of type $A \rightarrow B$ and a stream $xs$ of elements of type $A$ by applying the $n$th function in $fs$ to the $n$th value in $xs$. Such streams are thought of as values which vary in time.

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

$$
codata \Box A = \cdot : \cdot \ of \ A \times \Box \ A
$$

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

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

$$
\begin{align*}
\mathit{hd} & : \Box A \rightarrow A \\
\mathit{tl} & : \Box A \rightarrow \Box \ A
\end{align*}
$$

Finally, we can implement the $\mathit{app}$ function as follows:

$$
\begin{align*}
\mathit{app} & : \Box (A \rightarrow B) \rightarrow \Box A \rightarrow \Box B \\
\mathit{app} \; fs \; xs & = \\
\mathit{let} \ \bullet \ fs' = \mathit{tl} \; fs & \\
\ \bullet \ xs' = \mathit{tl} \; xs & \\
\in \ ((\mathit{hd} \; fs) \; (\mathit{hd} \; xs)) : : (\bullet \ (\mathit{app} \; fs' \; xs'))
\end{align*}
$$

We use the $\circ$ elimination form, $\mathit{let} \ \bullet$ to bind the variables $fs'$ and $xs'$ to values of the remaining streams in the next time step. Our typing rules will guarantee that $fs'$ and $xs'$ are only usable underneath $\bullet$, which will explain further in the following examples.

Such a program is interpreted reactively as a process which, at each time step, receives a function $A \rightarrow B$ and a value $A$ and produces a value $B$. More generally, given $n$ functions $A \rightarrow B$ 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 $\mathit{may}$ eventually occur and events that $\mathit{must}$ eventually occur. This is a feature not present in the line of work by Krishnaswami and his collaborators, Benton and Hoffman [24; 26] – they have $\mathit{unique}$ 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 $\emptyset$ operator (read: “eventually”) as a data type, corresponding to a type of events in reactive programming:

$$
data \emptyset A = \mathit{Now} \ of \ A \mid \mathit{Later} \ of \ \emptyset \ \emptyset A
$$

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

$$
\begin{align*}
\mathit{evapp} & : \emptyset A \rightarrow \Box (A \rightarrow B) \rightarrow \Box B \\
\mathit{evapp} \; ea \; fs & = \mathit{case} \; ea \; of \\
| \mathit{Now} \; x & \Rightarrow \mathit{Now} \; ((\mathit{hd} \; fs) \; x) \\
| \mathit{Later} \; ea' & \Rightarrow \mathit{let} \ \bullet \; ea'' = ea' \\
& \bullet \; fs' = \mathit{tl} \; fs \\
\in \ \mathit{Later} \ (\bullet \ (\mathit{evapp} \; ea'' \; fs'))
\end{align*}
$$

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 $\mathit{bool}$).

However, all functions with this type (in our system) have the property that given an $A$ is eventually provided, it $\mathit{must}$ eventually produce a $B$, although perhaps not at the same time. This is our first example of a $\mathit{liveness}$ 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 [19] 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:

$$
data \mathit{Req} = \mathit{Get} \mid \mathit{Put} \\
data \mathit{Resp} = \mathit{OK} \mid \mathit{Error}
$$

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):

$$
codata \mathit{Server} = \{ \mathit{noreq} : \emptyset \mathit{Server}, \mathit{some} : \mathit{Req} \rightarrow \mathit{Resp} \times \emptyset \mathit{Server} \}
$$

Now we can write the server program which responds to $\mathit{Get}$ with OK and Put with $\mathit{Error}$:

$$
\mathit{server} : \mathit{Server} \\
server = \{ \mathit{noreq} = \bullet \mathit{server}, \mathit{some} = \lambda r. \begin{cases} \mathit{OK} & \text{if } r \mathit{isGet} \ r \\
\mathit{else} & \mathit{Error} \end{cases} \}$$
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 \(\Box\) 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 \(A\):

\[
\text{predictor} : \Box A \rightarrow A
\]

\[
\text{predictor} \ x = \text{let} \ x' = x \text{ in } x'
\]

--- 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 [24].

Similarly, the type \(\Box (A + B)\) 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:

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

\[
\text{predictor} \ x = \text{let} \ x' = x \text{ in } \text{case } x' \text{ of}
\]

--- does not typecheck

| \(\text{inl} \ a\) ⇒ \text{inl} (\(\bullet\) \(a\)) |
| \(\text{inr} \ a\) ⇒ \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, \(\Box \otimes \rightarrow \bot\) is uninhabited in our system; one cannot get out of doing work today by citing the future.

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

\[
\text{import} : A \rightarrow \Box A
\]

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

This is disallowed on the grounds that it is not uniformly a theorem of LTL. Krishnaswami and Benton allow it in [25], but disallow it later in [26] 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:

\[
\text{import} : \text{Nat} \rightarrow \Box \text{Nat}
\]

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

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

Our language does not have Nakano’s guarded recursion [16]

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

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 \(\Box A\) in which the \(A\) is never delivered:

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

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

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.

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

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

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

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. [26] 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 \(A \mathcal{U} B\). This is a sequence of \(A\)s, terminated with a \(B\). In the setting of reactive programming, Jeltsch [21] calls programs of this type processes—they behave as time-varying signals which eventually terminate with a value.

\[
\text{data} \ A \mathcal{U} B = \text{Same} A \times (\Box (A \mathcal{U} B))
\]

\[
| \text{Switch of } B
\]

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, \(A \mathcal{U} B\) is isomorphic to \((\text{List } A) \times B\), but because of the \(\Box\), it matters when the \(B\) happens.

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

\[
\text{type} \ A \mathcal{U} B = A \times \Box (A \mathcal{U} B)
\]

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:

\[
\text{codata} \ \text{Fair } A B =
\]

\[
\text{In of } (A \mathcal{U} (B \mathcal{U} (A \times \Box (\text{Fair } A B))))
\]

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 \(B\):

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

\[
\text{sched as } bs =
\]
let \( n' = \text{import} \ n \)

- \( \text{as}' = \text{tl} \ \text{as} \)
- \( \text{bs}' = \text{tl} \ \text{bs} \)

\[
\text{let} \quad \text{sch2'} = \text{In} (\text{cnt} \ n \ n' \ \text{as} \ \text{bs})
\]

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 \text{import} for the corresponding stream types, but at the cost of a space leak.

We will need a special notion of “timed natural numbers” to implement the countdown. In the \text{import} case, the predecessor is only \text{tl} of \( \text{import} \ n' \) or \( \text{tl} \ \text{as}' \). Again the type requires us to eventually serve a \( B \).

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 \( B \).

\[
\text{data} \quad \text{TNat} = \text{Zero} | \text{Succ} (\text{TNat})
\]

We can write a function which imports \( \text{TNat}s \):

\[
\text{import} : \text{TNat} \to \text{TNat}
\]

\[
\text{import} \ n = \text{case} \ n \ of
\]

- \( \text{Zero} \Rightarrow \text{Zero} \)
- \( \text{Succ} \ p \Rightarrow \text{let} \quad \text{\( n' = p \) in}
\]

- \( \text{(\text{cnt} \ n' \ n' \ \text{as} \ \text{bs})} \)

- \( \text{sch2'} \quad \text{as} \quad \text{bs} = \text{sch2'} \ \text{Zero} \ \text{as bs} \)

\[
\text{let} \quad \text{sch2} : \text{TNat} \to \text{FA} \to \text{FB} \to \text{FA B}
\]

\[
\text{sch2} \ n \ a \ b = \text{sch2'} \ (\text{import} \ n) \text{as} \ \text{bs}
\]

\[
\quad \ldots \quad \text{Main function}
\]

\[
\text{sch2} : \text{FA} \to \text{FB} \to \text{FA B}
\]

\[
\text{sch2} \ n \ a \ b = \text{sch2'} \text{Zero} \text{as bs}
\]

\[
\text{Figure 2. Fair scheduler}
\]

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 \text{coiteration} operators available in the formal language, which we illustrate in the next section.

### 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 \text{if}operator for the corresponding stream types, but at the cost of a fixed points, \( \mu \) and \( \nu \) 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.

\[
\text{Types} \quad A, B, F, G ::= 1 | F \times G | F + G | A \to F | \quad F = \mu X.F | \nu X.F | X
\]

\[
\text{Terms} \quad M, N ::= x | (\) | (M, N) | \text{fst} \ M | \text{snd} \ M | \quad \text{inl} \ M | \text{inr} \ M | \text{case} \ M \text{of} \quad \text{inl} x \Rightarrow N | \text{inr} y \Rightarrow N' | \quad \lambda x.M | M \text{N} | M = M | \text{let} \quad \text{\( x = M \) in N} | \text{inj} M | \quad \text{iter}_x.\ F(x.M) | \text{out} \ M | \text{coi}\ F \ (x.M) | \text{map} (\Delta.\ F) \eta M
\]

\[
\text{Contexts} \quad \Theta, \Gamma ::= \cdot | \Gamma, x : A
\]

\[
\text{Kind Contexts} \quad \Delta ::= \cdot | \Delta, X : * \quad \text{Type substitutions} \quad \rho ::= \cdot | \rho, F/X
\]

\[
\text{Morphisms} \quad \eta ::= \cdot | \eta, (x.M)/X
\]

\[
\text{Figure 3. LTL Syntax}
\]

Our term language is mostly standard and we only discuss the terms related to \text{if} modality and the fixed points, \( \mu \) and \( \nu \). Let \( \Phi M \) describes a term \( M \) which is available in the next time step. Let \( \Phi x = M \) 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 guar-
antee that the variable only occurs under a •. Our language also includes iteration operator \textsc{iter}_X.F\ (x.M)\ N and coiteration operator \textsc{coiter}_X.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 \mu\ types is \textsc{inj}\ M, rolling up a term M. The elimination form for \nu\ types is out M, unrolling M.

We note that the X.F annotations on \textsc{iter} and \textsc{coiter} play a key role during runtime, since the operational semantics of map are defined using the structure of F (see Sec. 3). 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.:

\[\square A \equiv \nu X.A \times \bigcirc X\]

\[\blacklozenge A \equiv \mu X.A + \bigcirc X\]

The definition of \square A expresses that when we unfold a \square A, we obtain a value of type A (the head of the stream) and another stream in the next time step. The definition of \blacklozenge 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 great and least fixed points. Traditionally one expresses this by combining the always and eventually modalities, i.e. \square \blacklozenge A. However, there is another way to express this, namely:

\[\text{inf } A \equiv \nu X.\mu Y.(A \times \bigcirc X + \bigcirc Y)\]

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, \square \blacklozenge 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, \text{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}, A \cup B, A \cup B and \text{Fair } A B which we used in the examples in Sec. 2.

\[\begin{align*}
\text{Server} & \equiv \nu X.\bigcirc X \times (\text{Req} \rightarrow \text{Resp} \times \bigcirc X) \\
A \cup B & \equiv \mu X.(B + A \times \bigcirc X) \\
A \cap B & \equiv A \times \bigcirc (A \cap B) \\
\text{Fair } A B & \equiv \nu X.(A \cup B \cup (A \times \bigcirc X))
\end{align*}\]

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 \text{fs} and \text{x}s as the coinduction invariant:

\[\begin{align*}
\text{app} : \square(A \rightarrow B) \rightarrow \square A \rightarrow \square B \\
\text{app }fs\ x\ s\ s' \equiv \text{coiter}_X.B \times \bigcirc X\ (x.\ \text{let } \bullet\ x\ s' = \text{tl } (\text{fst } x)\ \text{in} \ \text{let } \bullet\ x\ s = \text{tl } (\text{snd } x)\ \text{in} \ ((\text{hd } (\text{fst } x)) \ (\text{hd } (\text{snd } x)), \bullet (fs', x\ s'))
\end{align*}\]

In the formal language, evapp becomes the following:

\[\begin{align*}
\text{evapp} : \square(A \rightarrow B) \rightarrow \square A \rightarrow \square B \\
\text{evapp }\equiv \text{let } \cdot x\ s\ s' = \text{tl } (\text{fst } x)\ \text{in} \ \text{let } \cdot x\ s = \text{tl } (\text{snd } x)\ \text{in} \ ((\text{hd } (\text{fst } x)) \ (\text{hd } (\text{snd } x)), \bullet (fs', x\ s'))
\end{align*}\]

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 [24].

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 •) 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 [23] or by using different judgments (i.e. \text{now}, later, stable) [24], 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 into 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 \bigcirc\ modality and fixed points \mu\ and \nu. In the rule \text{I}_\sigma, the introduction rule for \bigcirc\ with corresponding constructor •, 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,


### Typing Rules for \( \to, \times, +, \top, \bot \)

\[
\begin{align*}
\theta, \Gamma, x : A &\vdash M : B & \theta, \Gamma &\vdash M : A \to B & \theta, \Gamma &\vdash N : A \\
\theta, \Gamma &\vdash \lambda x. M : A \times B & \theta, \Gamma &\vdash M N : B \\
x : A \in \Gamma &\vdash \Theta, \Gamma &\vdash \cdot : 1 \\
\theta, \Gamma &\vdash M : A \times B & \theta, \Gamma &\vdash M : A \\
\theta, \Gamma &\vdash \text{fst} M : A & \theta, \Gamma &\vdash \text{snd} M : B \\
\theta, \Gamma &\vdash \text{in} j M : A + B & \theta, \Gamma &\vdash \text{in} N : A + B \\
\theta, \Gamma &\vdash M : A + B & \theta, \Gamma &\vdash x : A \to N_1 : C & \theta, \Gamma &\vdash y : B \to N_2 : C \\
\theta, \Gamma &\vdash \text{case} M : \text{in} l x \to N_1 \mid \text{in} r y \to N_2 : C \\
\end{align*}
\]

#### Rules for \( \bot \) modality and least and greatest fixed points

\[
\begin{align*}
\theta, \Gamma &\vdash M : A & \theta, \Gamma &\vdash M : \Theta; \Gamma \\
\theta, \Gamma &\vdash M : [\mu X.F/X] & \theta, \Gamma &\vdash \mu X.F : A \\
\theta, \Gamma &\vdash \text{in} j M : [\mu X.F/X] & \theta, \Gamma &\vdash \text{out} M : [\mu X.F/X] \\
\end{align*}
\]

#### Typing Rules for morphisms: \( \rho_1, \eta : \theta \to \Gamma \)

\[
\begin{align*}
\rho_1 &\vdash \eta : \rho_2 \\
\rho_1, A/X &\vdash \eta : (x.M)/X & \rho_2, B/X &\vdash \cdot : \\
\end{align*}
\]

**Figure 5. Typing Rules**


### 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 \( A \to \Box A \) (repeat), which is unsound for LTL and easily produces space leaks. Similarly, keeping \( \Gamma \) in the iter rule allows one to derive the unsound \( A \to \Box B \to A \lor B \) which says that if \( A \) holds now, and \( B \) eventually holds, then \( A \) in fact holds until \( B \) holds.

\[
\text{unsound} : A \to \Box B \to A \lor B \\
\text{unsound} \equiv \forall a. \lambda b. \text{iter}_X \cdot (b + \Box X) \ (y. \ case \ y \ of \ [\text{in} \ b] \to \text{in} \ (\text{in} \ b) \mid \text{in} \ u \to \text{in} \ (\text{in} \ (a, u)) - \alpha \text{ out of scope!}) \\
\]

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:

\[
\begin{align*}
F(\mu F) &\to F(C) & C &\to \nu F \\
\text{in} &\to \text{iter} f & f &\to F(\nu f) \\
\mu F &\to \text{map} (\Delta, F) \eta M & \eta M &\to \nu F \\
\theta, \Gamma &\vdash \text{rec} M \ (x.M) : N : C \\
\end{align*}
\]

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.

\[
\begin{align*}
\mu \vdash X. &\to \nu X \\
\frac{\mu X : \mu X.F/X &\vdash M : C}{\theta, \Gamma &\vdash \text{rec} M (x.M) : N : C} \\
\end{align*}
\]

The term 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 \( F(Y) = \text{list} \ Y = \mu X.1 + Y \times X. \) If \( \gamma : A \vdash M : B \) and \( N : \text{list} \ a, \text{then map} (Y, \text{list} Y) \ ((\gamma, M)/Y) : N : \text{list} B. \) 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 \([N/x]x^*M\). The key case in their definition is the case for \( \bullet M \). For current substitution \([N/x]x^*(\bullet M)\), in well-scoped terms, \( x \) cannot occur in \( M \), so we define:

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

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 \( \theta, \Gamma \vdash [N/x]M : B \)
\[ E_{k+1} := E_k \cdot \eta \] 
\[ E_k := [\cdot E_k \mid M E_k \mid fst E_k \mid snd E_k \mid (E_k, M) \mid (M, E_k) \mid \lambda x.E_k \mid case E_k \text{ of } \text{inl } x \rightarrow N \mid \text{inr } y \rightarrow N' \mid \text{inl } x \rightarrow E_k \mid \text{inr } y \rightarrow N \mid \text{inl } x \rightarrow N \mid \text{inr } y \rightarrow E_k \mid \text{let } \bullet x = E_k \text{ in } N \mid \text{let } \bullet x = M \text{ in } E_k \mid \text{inj } E_k \mid \text{out } E_k \mid \text{map } (\Delta, F) \eta E_k \]

**Figure 6. Evaluation contexts**

\[
\begin{aligned}
(\lambda x.M) N & \Rightarrow [N/x]M \\
stf(M, N) & \Rightarrow M \\
snd(M, N) & \Rightarrow N \\
\text{case } (\text{inl } M) \text{ of } \text{inl } x \mapsto N_1 \mid \text{inr } y \mapsto N_2 & \Rightarrow [M/x]N_1 \\
\text{case } (\text{inr } M) \text{ of } \text{inl } x \mapsto N_1 \mid \text{inr } y \mapsto N_2 & \Rightarrow [M/y]N_2 \\
\text{let } \bullet x = M \text{ in } N & \Rightarrow [\cdot M]N \\
\text{iter}_{X,F} (x. M) N \Rightarrow [\cdot \text{map } (X, F) (y. \text{iter}_{X,F} (x. M) y) / X] [N/x]M \\
\text{out } (\text{coit}_{X,F} (x. M) N) & \Rightarrow [\cdot \text{map } (X, F) (y. \text{coit}_{X,F} (x. M) y) / X] (\cdot [N/x]M) \\
\end{aligned}
\]

**Figure 7. Operational Semantics**

2. If \( \Theta; x : A; \Gamma \vdash M : B \) and \( ; \Theta \vdash N : A \) then \( ; \Theta; \Gamma \vdash [N/x] \cdot M : B \)

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 \( \Box \) 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 \( k \leq \alpha \) and \( M \) reduces to \( N \). More precisely, the reduction rule takes the following form:

\[
M \rightarrow N 
\]

\[ E_k[M] \rightarrow_{\alpha} E_k[N] \text{ if } k \leq \alpha \]

If \( \alpha = 0 \), then we are evaluating all redexes now and do not evaluate terms under a \( \Box \) 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 \( \bullet \).

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 coreursive value by \( \text{out } (\text{coit}_{X,F} (x. M) N) \), 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 \( \text{iter}_{X,F} (x. M) (\text{inj } N) \), 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 map, which witnesses that \( F \) is a functor. The operational semantics of 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 map in the syntax in the first place, are those for \( \mu \) and \( v \).

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: \( 0 \equiv \text{inj} (\text{inl}()) \), \( \text{suc } w = \text{inj} (\text{inr } w) \), \( 1 = \text{suc } 0 \), etc.

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

\[
\begin{aligned}
\text{double } 0 & \Rightarrow \text{db } (\text{inj} (\text{inl}())) \\
& \Rightarrow \text{case } M_0 \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\end{aligned}
\]

where

\[
\begin{aligned}
M_0 & = \text{map} (1 + X) (y. \text{db } y/X) \text{ (inl }()) \\
& \Rightarrow^* \text{case } (\text{inl }()) \text{ of } \text{inl } v \mapsto \text{inl }() \mid \text{inr } u \mapsto \text{inr } (\text{db } u) \\
& \Rightarrow^* \text{case } (\text{inr }()) \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w) \\
& \Rightarrow 0
\end{aligned}
\]

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

\[
\begin{aligned}
\text{double } 1 & \Rightarrow \text{db } (\text{inj} (\text{inr }0)) \\
& \Rightarrow \text{case } M_1 \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w)
\end{aligned}
\]

where

\[
\begin{aligned}
M_1 & = \text{map} (1 + X) (y. \text{db } y/X) \text{ (inr }0) \\
& \Rightarrow^* \text{case } (\text{inr }0) \text{ of } \text{inl } v \mapsto \text{inl }() \mid \text{inr } u \mapsto \text{inr } (\text{db } u) \\
& \Rightarrow^* \text{case } (\text{inr }0) \text{ of } \text{inl } y \mapsto 0 \mid \text{inr } w \mapsto \text{suc } (\text{suc } w) \\
& \Rightarrow \text{suc } (\text{suc }0) = 2
\end{aligned}
\]

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

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

Observe that after evaluating \( M \rightarrow_{\alpha} N \), where \( N \) is in normal form, one can then resume evaluation \( N \rightarrow_{\alpha+1} N' \) to obtain the culmulative 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 \( \rightarrow_{\alpha} \) (see below), however considering the general \( \rightarrow_{\alpha} \) 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 \( \leadsto \), which are important in our proof of strong normalization. Notice that these are independent of any typing assumptions.

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

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 [14, 15, 36]. In our setting, this means we prove that for any \( \alpha \leq \omega \), every reduction sequence \( M \leadsto \ast \) ... 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 suitable causal using normalization at 0, 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 \( \alpha \leq \omega \) to keep track of how many time steps we are normalizing. It is degenerate in that the partial order we use on \( \omega \) is discrete, i.e. \( \alpha \leq \beta \) precisely when \( \alpha = \beta \). It is not step-indexed because our strict positivity condition on \( \mu \) and \( \nu \) means we can interpret them without the aid of step indexing.

This proof is made challenging by the presence of interleaving \( \mu \) and \( \nu \) types and (co)iteration operators. This is uncommon, and has been treated before by Matthes [29] as well as Abel and Altenkirch [2]. Others have considered non-interleaving cases or with other forms of recursion, such as Mendler [30] and Jouannaud and Okada [22]. To our knowledge, ours is the first such proof which treats map as primitive syntax with reduction rules instead of a derivable map \( \eta \). To our knowledge, ours is the first such proof which treats map as primitive syntax with reduction rules instead of a derivable map \( \eta \).

We use a standard inductive characterization of strongly normalizing:

**Definition 4** (Strongly normalizing). We define \( \text{sn} \) as the inductive closure of:

\[ \forall M', M \leadsto \ast M' \implies M' \in \text{sn}_\alpha \]

We say \( M \) is strongly normalizing at \( \alpha \) if \( M \in \text{sn}_\alpha \).

It is immediate from this definition that if \( M \in \text{sn}_\alpha \) and \( M \leadsto \ast M' \) then \( M' \in \text{sn}_\alpha \). Since this is an inductive definition, it affords us a corresponding induction principle: To show that a property \( P \) holds of a term \( M \in \text{sn}_\alpha \), one is allowed to assume that \( P \) holds for all \( M' \) such that \( M \leadsto \ast M' \). One can easily verify by induction that if \( M \in \text{sn}_\alpha \) then there are no infinite \( \leadsto \) reduction sequences rooted at \( M \).

For our proof, we use Tait's saturated sets instead of Girard's reducibility candidates, as this allows us to perform the syntactic analysis of redexes separate from the more semantic parts of the proof. This technique has been used by Luo [23, Altenkirch [4], and Matthes [29].

In the following, we will speak of indexed sets, by which we mean a subset of terms for each \( \alpha \leq \omega \), i.e. \( A : \omega + 1 \to P(\text{tm}) \), where we write \( \text{tm} \) for the set of all terms. We overload the notation \( \subseteq, \cap, \cup \) to mean pointwise inclusion, intersection, and union.

That is, if \( A \) and \( B \) are indexed sets, we will write \( A \subseteq B \) to mean \( A^\alpha \subseteq B^\alpha \) for all \( \alpha \).

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

\[
[\bullet A]_0^\alpha \equiv \text{tm} \\
[\bullet A]_{m+1}^\alpha \equiv A_m \\
[\bullet A]^\omega \equiv A^\omega
\]

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 \( \bullet \).

In Figure 9 we define a notion of normalizing weak head reduction. We write \( M \leadsto \ast M' \) for a contraction, and \( M \leadsto \ast \ast N \) 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.

\[
\mathcal{H} ::= \emptyset | \text{fst} \mathcal{H} | \text{snd} \mathcal{H} | \mathcal{H} N | \text{iter}_x.F(x,M) \mathcal{H} | \text{map} (\Delta \mu X.F) \eta \mathcal{H} | \text{out} \mathcal{H} \quad \text{let} \quad x = \mathcal{H} \quad \text{in} \quad N \quad \text{case} \quad \mathcal{H} \quad \text{of} \quad \text{inl} \ x \mapsto N_1 \quad \text{inr} \ y \mapsto N_2
\]

\[
M \leadsto \ast M' \quad \mathcal{H}[M] \leadsto \ast \mathcal{H}[M']
\]

**Lemma 6.** \( \text{sn}_\alpha \) is backward closed under normalizing weak head reduction: If \( M' \in \text{sn}_\alpha \) and \( M \leadsto \ast M' \) then \( M \in \text{sn}_\alpha \).
Proof. We consider each redex in the definition of $\Rightarrow$. The proofs are by lexicographic induction on the derivations of $\mathit{sn}_\alpha$. We show some of the cases of interest:

(1) We show by lexicographic induction, first on $\mathit{N} \in \mathit{sn}_\alpha$ and second on $\mathcal{H}(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ that $\mathcal{H}((\mathit{lambda}M.N)[M]) \in \mathit{sn}_\alpha$.

We must show that every $M'$ such that $\mathcal{H}((\mathit{lambda}M.N)[M']) \in \mathit{sn}_\alpha$ is in $\mathit{sn}_\alpha$. So we analyze the ways $\mathcal{H}((\mathit{lambda}M.N))$ can reduce:

(a) $\mathcal{H} \Rightarrow \mathcal{H}'$: then $\mathcal{H}(\mathit{N}/x)[M] \Rightarrow \mathcal{H}'(\mathit{N}/x)[M]$, and hence by the induction hypothesis with the smaller derivation $\mathcal{H}'(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ we have $\mathcal{H}(\mathit{lambda}M.N)[M] \in \mathit{sn}_\alpha$ as required.

(b) $M \Rightarrow M'$: then $\mathcal{H}(\mathit{N}/x)[M'] \Rightarrow \mathcal{H}'(\mathit{N}/x)[M']$, and hence by the induction hypothesis with the smaller derivation $\mathcal{H}'(\mathit{N}/x)[M'] \in \mathit{sn}_\alpha$ we have $\mathcal{H}(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ as required.

(c) $N \Rightarrow N'$: then $\mathcal{H}(\mathit{N}/x)[M] \Rightarrow \mathcal{H}'(\mathit{N}/x)[M]$, and hence $\mathcal{H}(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ (although this derivation is not necessarily smaller, since it may have taken 0 steps). By the induction hypothesis with the smaller derivation $N' \in \mathit{sn}_\alpha$, we have $\mathcal{H}(\mathit{lambda}M.N)[N'] \in \mathit{sn}_\alpha$ as required.

(d) $(\mathit{lambda}M)[N] \Rightarrow (\mathit{N}/x)[M]$: we have $\mathcal{H}(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ by assumption.

By careful inspection of the reduction rules, no other cases are possible.

(4) We show by lexicographic induction, first on $\mathit{N} \in \mathit{sn}_\alpha$ and second on $\mathcal{H}(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ that $\mathcal{H}((\mathit{let} \bullet x \Rightarrow \bullet N)[M]) \in \mathit{sn}_\alpha$.

We analyze the ways $\mathcal{H}((\mathit{let} \bullet x \Rightarrow \bullet N)[M])$ can reduce:

(a) $\mathcal{H} \Rightarrow \mathcal{H}'$: then $\mathcal{H}(\mathit{N}/x)[M] \Rightarrow \mathcal{H}'(\mathit{N}/x)[M]$, and hence by the induction hypothesis with the smaller derivation $\mathcal{H}'(\mathit{N}/x)[M] \in \mathit{sn}_\alpha$ we have $\mathcal{H}((\mathit{let} \bullet x \Rightarrow \bullet N)[M]) \in \mathit{sn}_\alpha$ as required.

(b) $M \Rightarrow M'$: then $\mathcal{H}(\mathit{N}/x)[M] \Rightarrow \mathcal{H}(\mathit{N}/x)[M']$, and hence by the induction hypothesis with the smaller derivation $\mathcal{H}(\mathit{N}/x)[M'] \in \mathit{sn}_\alpha$, we have $\mathcal{H}(\mathit{let} \bullet x \Rightarrow \bullet N)[M'] \in \mathit{sn}_\alpha$ as required.

\[ \mathcal{H}(\mathit{N}/x)[M] \rightarrow^{\alpha} \mathcal{H}(\mathit{N}/x)[M'] \in \mathit{sn}_\alpha, \] we have $\mathcal{H}(\mathit{let} \bullet x \Rightarrow \bullet N)[M'] \in \mathit{sn}_\alpha$ as required.

(c) $\mathit{let} \bullet x \Rightarrow \bullet N' \Rightarrow \bullet N'$: then $\mathcal{H}(\mathit{N}/x)[M] \Rightarrow \mathcal{H}(\mathit{N}/x)[M']$, and hence by the induction hypothesis with the smaller derivation $\mathcal{H}(\mathit{N}/x)[M'] \in \mathit{sn}_\alpha$, we have $\mathcal{H}(\mathit{let} \bullet x \Rightarrow \bullet N)[M] \in \mathit{sn}_\alpha$ as required.

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 $\Rightarrow$. For this reason, we typically highlight the $\Rightarrow$ cases of the proofs.

We define the indexed set of terms $\mathit{sne}$ ("strongly normalizing neutral") which are strongly normalizing, but are stuck with a variable in place of a weak head redex.

\begin{definition} We define $\mathit{sne}_\alpha = \{ \mathit{H}[x] \in \mathit{sn}_\alpha \}$ \end{definition}

Lemma 8. The rules presented in Figure 7 are admissible for $\mathit{sn}$ and $\mathit{sne}$.

**Proof.** We show the case $\mathit{let} \bullet x \subseteq \mathit{sn} \circ \bullet$:

Suppose $M \in \mathit{sn}_\alpha$. Case $\alpha = 0$: then $M \in \mathit{sn}_0$ because there are no $M'$ such that $M \Rightarrow^{\alpha} M'$.

Case $\alpha = m + 1$: then $M \in \mathit{sn}_m$. We show by induction on the derivation of $M \in \mathit{sn}_m$ that $\bullet M \in \mathit{sn}_{m+1}$:

Figure 9. Normalizing weak head reduction
Suppose $M \leadsto_{m+1} N$. By inversion, we have $N = P M'$ and $M \leadsto_{m} M'$. By I.H., $M' \in sn_{m+1}$ as required.

Case $\alpha = \omega$. Then $M \in sn_{\omega}$. We show by induction on the derivation of $M \in sn_{\omega}$ that $\bullet M \in sn_{\omega}$:

1. $M \leadsto_{\omega} N$. By inversion, we have $N = P M'$ and $M \leadsto_{\omega} M'$. By I.H., $\bullet M' \in sn_{\omega}$ as required.

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

1. $A \subseteq sn$
2. For any $\alpha, M, M'$, if $M \in \nu_{\alpha}^* M' \rightarrow_{\alpha} M'$ then $M \in A^\alpha$ (Backward closure under normalizing weak head reduction)
3. $\text{sne} \subseteq A$

It is immediate from Lemma 8 that $sn$ is saturated.

**Definition 10**. For an indexed set $A : \omega + 1 \rightarrow \mathcal{P}(tm)$, we define $\overline{A}$ as its closure under conditions 2 and 3. i.e.

$$\overline{A}^\alpha \equiv \{ M \in \overline{\nu_{\alpha}^* M'} \rightarrow_{\alpha} M' \land (M' \in A^\alpha \lor M' \in \text{sne}_{\alpha}) \}$$

**Lemma 11**. We have the following properties of $(\overline{-})$:

1. $\overline{\overline{P}} = P$
2. $P \subseteq \overline{P}$
3. If $A \subseteq B$ then $\overline{A} \subseteq \overline{B}$ (monotonicity)
4. If $A$ is a set and $B$ is saturated with $A \subseteq B$ then $\overline{A} \subseteq \overline{B}$ (adjunction *)
5. If $A$ is an indexed set such that $A \subseteq sn$, then $\overline{A}$ is saturated.

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

**Lemma 12**. Saturated sets form a complete lattice under $\subseteq$, with greatest lower bounds and least upper bounds given by:

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

For $\land$, we intersect with $\text{sn}$ so that the nullary lower bound is $sn$, and hence 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:

**Corollary 13**. Given $F$ which takes predicates to predicates, we define:

$$\mu F \equiv \bigvee \{ C \subseteq A \mid F(C) \subseteq C \}$$

$$\nu F \equiv \bigwedge \{ C \subseteq A \mid C \subseteq F(C) \}$$

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.

In what follows, we will use the characterization of least fixed points as least pre-fixed points (and correspondingly for greatest fixed points):

**Property 14**. If $F$ is a monotone operator on saturated sets and $C$ is saturated, then:

1. $F(\mu F) \subseteq \mu F$
2. If $F(C) \subseteq C$ then $\mu F \subseteq C$ ($\mu$ induction)
3. $\nu F \subseteq F(\nu F)$
4. If $\overline{C} \subseteq F(C)$ then $C \subseteq \nu F$ ($\nu$ coinduction)

**Lemma 15**. If $F$ is monotone in its $n + 1$ arguments, then $\rho \mapsto \mu (X \mapsto F(\rho, X))$ is monotone in its $n$ arguments. Dually, so is $\rho \mapsto \nu (X \mapsto F(\rho, X))$.

**Proof**. Suppose $\rho_1 \subseteq \rho_2$ (pointwise).

1. $\rho_1, \mu (X \mapsto F(\rho_2, X)) \subseteq \rho_2, \mu (X \mapsto F(\rho_2, X))$ (pointwise)
2. $\rho_2, \mu (X \mapsto F(\rho_2, X)) \subseteq F(\rho_2, \mu (X \mapsto F(\rho_2, X)))$ (monotonicity of $F$)
3. $\rho_2, \mu (X \mapsto F(\rho_1, X)) \subseteq \mu (X \mapsto F(\rho_2, X))$ (pre-fixed point)

The following 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 16**. We define:

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

$$(A \times f)^\alpha \equiv \{ f M \mid M \in A^\alpha \}$$

We will often use these notations for partially applied syntactic forms, e.g. $A^\alpha \circ f$

**Lemma 17**. The following properties hold:

1. For any $f$, if $A \subseteq B$ then $A \circ f \subseteq B \circ f$ (monotonicity)
2. For any $f$, if $A \subseteq B$ then $A \times f \subseteq B \times f$ (monotonicity)
3. For any $f$ and set $A$, we have $A \subseteq (A \circ f) \circ f$
4. For any $f$ and sets $A, B$, we have $A \times f \subseteq B$ if and only if $A \subseteq B \circ f$ (Adjunction *)

**Lemma 18** (Subterm property of $sn$). If $\mathcal{H}[M] \in sn_{\alpha}$ where $\mathcal{H}$ is a head context, then $M \in sn_{\alpha}$

**Proof**. By induction on the derivation $\mathcal{H}[M] \in sn_{\alpha}$.

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

**Definition 20**. We define the following operations on saturated sets:

1. $\text{sn}$
2. If $\mathcal{H'}[M] \leadsto_{\alpha} \mathcal{H}[M]$ and $\mathcal{H}[M'] \in (A \circ \mathcal{H})^\alpha$ then $\mathcal{H}'[\mathcal{H}[M]] \leadsto_{\alpha} \mathcal{H}[\mathcal{H}[M']]$ and $\mathcal{H}'[\mathcal{H}[M']] \in \mathcal{A}^\alpha$, hence $\mathcal{H}'[\mathcal{H}[M]] \in \mathcal{A}^\alpha$ (by closedness of $A$).
3. $\text{sne} \subseteq \text{sne} \circ \mathcal{H} \subseteq A \circ \mathcal{H}$ by lemmas 18,17

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 21.** The operators defined in Definition 20 take saturated sets to saturated sets.

**Proof.** For 1, ×, µ, ν observe that previous lemmas establish that the operations involved in their definitions take saturated sets to saturated sets.

Case \(A + B\): Suppose \(A\) and \(B\) are saturated.

1. \(A \subseteq s\) (since \(A\) saturated)
   \[\subseteq s n \circ \text{inl} \quad \text{(by lemma 8)}\]
2. \(A \ast \text{inl} \subseteq s\) (by lemma 17)
3. \(B \ast \text{inr} \subseteq s\) (similarly)
4. \((A \ast \text{inl}) \cup (B \ast \text{inr}) \subseteq s\) (by lemma 11)
5. \((A \ast \text{inl}) \cup (B \ast \text{inr})\) is saturated (by lemma 11)

Case \(A \rightarrow B\): Suppose \(A\) is saturated.

1. Suppose \(M \in (A \rightarrow B)^\alpha\), i.e. for any \(N \in A^\alpha\) we have \(M \cdot N \in B^\beta\).
   (a) \(x \in A^\alpha\) (since \(A\) is saturated)
   (b) \(M \cdot x \in B^\beta\) (by assumption)
   (c) \(M \cdot x \in sn^\alpha\) (since \(B\) is saturated)
   (d) \(M \cdot x \in sn^\alpha\) (by lemma 18)
   So \((A \rightarrow B) \subseteq s\)
2. Suppose \(M \rightarrow_\alpha M'\) with \(M' \in (A \rightarrow B)^\alpha\), i.e. for any \(N \in A^\alpha\) we have \(M' \cdot N \in B^\beta\). Suppose \(N \in A^\alpha\).
   (a) \(M \cdot N \rightarrow_\alpha M' \cdot N\) (by def of \(\rightarrow_\alpha\))
   (b) \(M \cdot N \in B^\beta\) (since \(B\) is saturated)
   So \(A \rightarrow B\) is backward closed under \(\rightarrow_\alpha\).
3. Suppose \(M \in sne_{\alpha}\). Suppose \(N \in A^\alpha\). Then \(M \cdot N \in sne_{\alpha}\), hence \(M \cdot N \in B^\beta\) (since \(B\) is saturated). Hence \(M \in (A \rightarrow B)\)

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

**Definition 22.** 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:

\[
\begin{align*}
X & \mapsto \rho(X) \\
1 & \mapsto 1 \\
F \times G & \mapsto [F](\rho) \times [G](\rho) \\
F + G & \mapsto [F](\rho) + [G](\rho) \\
A \rightarrow F & \mapsto [A](\rho) \rightarrow [F](\rho) \\
\text{coit}_{X,F} & \mapsto (\Delta, \nu X.F) \quad \eta N \in sne_{\alpha}
\end{align*}
\]

Observe that, by lemma 22 every \([F](\rho)\) is saturated.

**Lemma 23** (Monotonicity of \([F]\)). If \(\rho_1 \subseteq \rho_2\) (pointwise)

And \(F\) is a well-formed type whose free variables match the domains of \(\rho_1\) and \(\rho_2\), then

\([F](\rho_1) \subseteq [F](\rho_2)\)

**Proof.** Observe that \(A \rightarrow (\_\_\_)\) is a monotone operator.

We have by now shown that every operator involved in the definition is monotone.

**Lemma 24** (\([-\_\_]\) is compositional). If \(\sigma\) is a substitution of (possibly open) types for types, and \(F\) is an open type, then:

\([\sigma][F](\rho) = [F](\sigma)(\rho)\)

If \(\theta = N_1/y_1, \ldots, N_n/y_n\) and \(\sigma = M_1/x_1, \ldots, M_m/x_m\) are simultaneous substitutions, we write \([\theta; \sigma][M]\) to mean substituting the \(N_i\) with next substitution \([N_i/y_i]\)(\_\_) and the \(M_i\) with the current substitution \([M_i/x_i]\)(\_\_), where \(i\) is positive.

**Definition 25** (Semantic typing). We write \(\Theta; \Gamma \vdash M : C\), where the free variables of \(M\) are bound by \(\Theta\) and \(\Gamma\), if for any \(\alpha\) and any substitutions \(\theta \in (\_\_\_\_\_)\) and \(\sigma \in (\_\_\_\_\_)\) we have \([\theta; \sigma][M] \in C^\alpha\)

**Lemma 26** (Semantic substitution lemma). If \(\Theta; \Gamma, x : A \vdash M : B\) and \(\Theta; \Gamma \vdash N : A\) then \(\Theta; \Gamma \vdash [N/x]M : B\)

**Proof.**
1. For any $\alpha$, substitutions $\theta \in (\Theta)^*\sigma \in \Gamma^\alpha$ and $N' \in A^\alpha$ we have $[\theta; \sigma, N'/x]M \in B^\alpha$ (assumption)

2. For any $\alpha$, and $\theta \in (\Theta)^*, \sigma \in \Gamma^\alpha$ we have $[\theta; \sigma]N \in A^\alpha$ (assumption)

3. So taking $N' = [\theta; \sigma]N$ in line 1, we have $[\theta; \sigma, [\theta; \sigma]N/x]M \in B^\alpha$.

4. Hence $[\theta; \sigma]([N/x]M) \in B^\alpha$ (since $[\theta; \sigma, [\theta; \sigma]N/x]M = [\theta; \sigma]([N/x]M)$

\[ \square \]

**Lemma 27.** If $\Theta, \Gamma, x : A \models H[x] : C$ where $C$ is saturated and $H$ is a head containing $x$, then $\Theta, \Gamma, x : A \models H[x] : C$.

Proof. We are given that for any $\alpha, \theta \in (\Theta)^*, \sigma \in \Gamma^\alpha$, $N \in A^\alpha$, we have $[\theta; \sigma, N/x]H[x] \in C$.

Notice $[\theta; \sigma, N/x]H[x] = ([\theta; \sigma]H)[N]$, and so our assumption states:

\[ A^\alpha \subseteq \{C \circ ([\theta; \sigma]H)\}^\alpha \]

By definition, this is what we need to prove.

\[ \square \]

**Lemma 28.** If $A$ is saturated and $\Theta, \Gamma \models M' : A$ and $\forall \alpha, \theta \in (\Theta)^*, \sigma \in \Gamma^\alpha$ we have $[\theta; \sigma]M \rightarrow^\alpha [\theta; \sigma]M'$ then $\Theta, \Gamma \models M : A$.

Next we show that the term constructors of our language obey corresponding semantic typing lemmas. We prove the easier cases first (i.e. constructors other than map, iter, and coit), as their results are implications in the next lemma pertaining to map.

**Lemma 29 (Interpretation of term constructors). The following hold, where we assume $\Theta, \Gamma, A, B, C$ are saturated.**

1. $\Theta, \Gamma, \{\} \models 1$
2. If $\Theta, \Gamma \models M : A$ and $\Theta, \Gamma \models N : B$ then $\Theta, \Gamma \models \{M, N\} : A \times B$
3. If $\Theta, \Gamma \models M : A \times B$ then $\Theta, \Gamma \models \fst\ M : A$
4. If $\Theta, \Gamma \models M : A \times B$ then $\Theta, \Gamma \models \snd\ M : B$
5. If $\Theta, \Gamma \models M \in A$ then $\Theta, \Gamma \models \in\ M : A + B$
6. If $\Theta, \Gamma \models M : B$ then $\Theta, \Gamma \models \in\ M : A + B$
7. If $\Theta, \Gamma \models M : A + B$ and $\Theta, \Gamma, x : A \models N_1 : C$ and $\Theta, \Gamma, y : B \models N_2 : C$ then $\Theta, \Gamma \models \case\ M : \case\ x y N_1 N_2 : C$
8. If $\Theta, \Gamma \models M \in A$ then $\Theta, \Gamma \models \case\ M : A \rightarrow B$
9. If $\Theta, \Gamma \models N \in A$ then $\Theta, \Gamma \models \case\ N : B$
10. If $\Theta, \Gamma \models M : A$ then $\Theta, \Gamma \models \case\ M : \case\ N A$
11. If $\Theta, \Gamma \models M \in \case\ N A$ and $\Theta, \Gamma \models N \in B$ then $\Theta, \Gamma \models \case\ N B : B$
12. If $F$ is a monotone function from saturated sets to saturated sets, and $\Theta, \Gamma \models M : F(\mu F)$ then $\Theta, \Gamma \models \case\ M : \mu F$
13. If $F$ is a monotone function from saturated sets to saturated sets, and $\Theta, \Gamma \models M : \nu F$ then $\Theta, \Gamma \models \case\ M : \nu F$

Proof. We show some representative cases. The others are similar.

2. (a) $\Theta, \Gamma \models M : A$ (assumption)
   (b) $\Theta, \Gamma \models N : B$ (assumption)
   (c) $\Theta, \Gamma \models \fst\ M, N : A$ (by $\rightarrow\)$ closure
   (d) $\Theta, \Gamma \models (M, N) : A \circ \fst$
   (e) $\Theta, \Gamma \models \snd\ M, N : B$ (by $\rightarrow\)$ closure, line (b))
   (f) $\Theta, \Gamma \models (M, N) : B \circ \snd$
   (g) $\Theta, \Gamma \models (M, N) : A \circ \fst\ \circ \snd\$ (lines (d),(f))
3. (a) $\Theta, \Gamma, x : A \models N_1 : A$ (assumption)
   (b) $\Theta, \Gamma, x : B \models N_2 : B$ (assumption)

(c) $\Theta, \Gamma, x : A \models \case\ x y N_1 N_2 : C$ (by $\rightarrow\)$ closure
(d) $\Theta, \Gamma, z : A \times B \models \case\ z y N_1 N_2 : C$
(e) $\Theta, \Gamma, x : B \models \case\ x y N_1 N_2 : C$
(f) $\Theta, \Gamma, z : B \times B \models \case\ z y N_1 N_2 : C$
(g) $\Theta, \Gamma, z : (A \times B + B \times B) \models \case\ z y N_1 N_2 : C$

(h) $\Theta, \Gamma, z : (\ case\ z y N_1 N_2 : C$

(i) $\Theta, \Gamma \models M : A + B$ (assumption)
(j) $\Theta, \Gamma \models \case\ M : \case\ y N_1 N_2 : C$ (by def)

(k) $\Theta, \Gamma \models M : \case\ M : A$ (assumption)

(l) $\Theta, \Gamma \models \case\ M : B : C$ (by def)

\[ \square \]
4. \( x : [G](\rho_1) \vdash \text{map } G \eta : [G](\rho_2) \) (by I.H.)
5. \( y : [F](\rho_1) \times [G](\rho_1) \vdash \text{snd } y : [G](\rho_1) \) (by lemma 29)
6. \( y : [F](\rho_1) \times [G](\rho_1) \vdash \text{map } G \eta \text{ (snd } y) : [G](\rho_2) \) (by sem. subst. 26)
7. \( y : [F](\rho_1) \times [G](\rho_1) \vdash (\text{map } F \eta \text{ (fst } y), \text{map } G \eta \text{ (snd } y)) : [F](\rho_2) \times [G](\rho_2) \) (by lemma 29, lines 3, 6)
8. \( y : [F](\rho_1) \times [G](\rho_1) \vdash \text{map } (F \times G) \eta \text{ } y : [F](\rho_2) \times [G](\rho_2) \) (by \( \eta \text{ } y \))
9. \( y : [F \times G](\rho_1) \vdash \text{map } (F \times G) \eta \text{ } y : [F \times G](\rho_2) \) (by def)

Case \( A \rightarrow F \):
1. \( x : [F](\rho_1) \vdash \text{map } F \eta \text{ } x : [F](\rho_2) \) (by I.H.)
2. \( y : [A](\cdot) \rightarrow [F](\rho_1), z : [A] \vdash y \text{ } z : [F](\rho_1) \) (by lemma 29)
3. \( y : [A](\cdot) \rightarrow [F](\rho_1), z : [A] \vdash \text{map } F \eta \text{ } (y \text{ } z) : [F](\rho_2) \) (by sem. subst. 26)
4. \( y : [A](\cdot) \rightarrow [F](\rho_1) \vdash \lambda z. \text{map } F \eta \text{ } (y \text{ } z) : [A](\cdot) \rightarrow [F](\rho_2) \) (by lemma 29)
5. \( y : [A](\cdot) \rightarrow [F](\rho_1) \vdash \text{map } (A \rightarrow F) \eta \text{ } y : [A](\cdot) \rightarrow [F](\rho_2) \) (by \( \eta \text{ } y \))
6. \( y : [A \rightarrow F](\rho_1) \vdash \text{map } (A \rightarrow F) \eta \text{ } y : [A \rightarrow F](\rho_2) \) (by def)

Case \( \bigcirc F \):
1. \( x : [F](\rho_1) \vdash \text{map } F \eta \text{ } x : [F](\rho_2) \) (by I.H.)
2. \( x : \bigcirc ([F](\rho_1)) \vdash (\text{map } F \eta \text{ } x) : \bigcirc ([F](\rho_2)) \) (by lemma 29)
3. \( y : \bigcirc ([F](\rho_1)) \vdash \text{let } x = y \text{ } in (\text{map } F \eta \text{ } x) : \bigcirc ([F](\rho_2)) \) (by lemma 29)
4. \( y : \bigcirc [F](\rho_1) \vdash \text{let } x = y \text{ } in (\text{map } F \eta \text{ } x) : \bigcirc [F](\rho_2) \) (by def)
5. \( y : \bigcirc [F](\rho_1) \vdash \text{map } \bigcirc (F \eta \text{ } y) : \bigcirc [F](\rho_2) \) (by \( \eta \text{ } y \))

Case \( \mu X.F \): This proceeds basically using the least fixed point property and → closure.
Let \( C = [\mu X.F](\rho_2) = \mu (X \rightarrow [F](\rho_2, X) \times \text{inj}) \)
Let \( D = C \circ (\mu (X.F) \eta) \)
Let \( N = (\text{map } F \eta, \text{map } (\mu X.F) \eta) \)

1. \( D \) is saturated (by Lemma 19)
2. \( x : D \vdash (\text{map } (X.F) \eta) \text{ } x : C \) (by definition of \( \vdash \), \( D \))
3. \( x : [F](\rho_1, D) \vdash \text{map } (\eta, \text{map } (\mu X.F) \eta) \text{ } x : [F](\rho_2, C) \) (by I.H.)
4. \([F](\rho_1, D) \subseteq [F](\rho_2, C) \circ (\text{map } F \eta, \text{map } (\mu X.F) \eta) \) (by definition of \( \subseteq \))
\( \leq [F](\rho_2, C) \times \text{inj} \times \text{inj} \times N \) \( \leq [F](\rho_2, C) \times \text{inj} \times \text{inj} \times N \) (by mon. of \( \circ \))
\( = C \circ \text{inj} \times N \) (by rolling fixed point)
\( = C \circ \text{inj} \times (\text{map } F \eta, \text{map } (\mu X.F) \eta) \) (by def)
\( \subseteq C \circ \text{inj} \times (\text{map } F \eta, \text{map } (\mu X.F) \eta) \) (by \( \eta \text{ } y \))
\( = C \circ \text{inj} \times N \) (by \( \eta \text{ } y \))
\( = C \circ \text{inj} \times \text{inj} \) (by \( \eta \text{ } y \))
\( = D \circ \text{inj} \) (by def)
5. \( [F](\rho_1, D) \times \text{inj} \subseteq D \) (by adjunction \( \times \))
6. \( [F](\rho_1, D) \times \text{inj} \subseteq D \) (by adjunction \( \langle - \rangle \))
7. \( [\mu X.F](\rho_1) \subseteq D \) (by \( \eta \text{ } y \))
8. \( y : [\mu X.F](\rho_1) \vdash (\text{map } (\mu X.F) \eta) \text{ } y : [\mu X.F](\rho_2) \) (by definitions of \( \vdash \), \( D \), \( C \))

Case \( \nu X.F \):
Let \( C = [\nu X.F](\rho_1) = \nu (X \rightarrow [F](\rho_1, X) \times \text{out}) \)
Let \( N = \text{map } (\nu X.F) \eta \)
Let \( D = C \times N \)

1. \( D \) is saturated (property of \( \langle - \rangle \), closure property of \( \text{sn} \))

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 →, appealing to the map lemma.

**Lemma 31.** If \( x : [F](\langle C \mid X \rangle) \vdash M : C \) where \( C \) is saturated, we have \( y : [\mu X.F] \vdash \text{iter}_X F (x.M) y : C \)

**Proof.** Let \( D = C \circ \text{iter}_X F (x.M) \)
1. \( D \) is saturated (by Lemma 19)
2. \( [F](\langle C \mid X \rangle) \subseteq C \circ (\langle - \mid X \rangle M) \) (by assumption)
3. \( D \subseteq C \circ \text{iter}_X F (x.M) \) (by def)
4. \( y : D \vdash \text{iter}_X F (x.M) y : C \) (by def of \( \vdash \))
5. \( y : [F](\langle D \mid X \rangle) \vdash \text{map } (X.F) \text{ } \eta : [F](\langle C \mid X \rangle) \)
6. \( [F](\langle D \rangle) \subseteq \text{map } (X.F) \text{ } \eta : [X.F](\langle \text{iter}_X F (x.M) \rangle) \) (by semantic typing of map, previous line)
7. \( [F](\langle D \rangle) \subseteq \text{sn} \times C \circ (\text{map } (X.F) \text{ } \eta : [\text{iter}_X F (x.M)]) \) (previous line, saturated)
8. \( [F](\langle D \rangle) \times \text{inj} \subseteq D \) (by adjunction \( \times \))
9. \( [F](\langle D \rangle) \times \text{inj} \subseteq D \) (by adjunction \( \langle - \rangle \))
10. \( \mu (X \rightarrow [F](\langle X \rangle) \times \text{inj}) \subseteq D \) (lfp property)
11. \( y : [\mu X.F] \vdash \text{iter}_X F (x.M) y : C \) (by definitions of \( \vdash \))

**Corollary 32.** If \( x : [F](\langle C \mid X \rangle) \vdash M : C \) and \( \Theta : \Gamma \vdash N : [\mu X.F] \) where \( C \) is saturated, then \( \Theta : \Gamma \vdash \text{iter}_X F (x.M) N : C \)

**Lemma 33.** If \( x : \Gamma \vdash M : [F](\langle C \mid X \rangle) \) where \( C \) is saturated, we have \( y : \Gamma \vdash \text{coit}_X F (x.M) y : [\nu X.F] \)

**Proof.** Let \( D = C \times (\text{coit}_X F (x.M)) \)
1. \( D \) is saturated (property of \( \langle - \rangle \), closure property of \( \text{sn} \))
2. $\mathcal{C} \subseteq \mathcal{C} \star (\text{coi}t_{X.F} (x.M)) \circ (\text{coi}t_{X.F} (x.M))$
\quad $\subseteq D \circ (\text{coi}t_{X.F} (x.M))$ (monotonicity of $\circ$)
3. $y : C \Rightarrow \text{coi}t_{X.F} (x.M) : D$ (by def of $\Rightarrow$)
4. $y : [F]([D]) \Rightarrow \text{map}(X.F)(\text{coi}t_{X.F} (x.M)) : [F][D]$ (semant
\quad ic typing of map)
5. $[F]([C]) \subseteq [F][D] \circ (\text{map}(X.F)(y \cdot \text{coi}t_{X.F} (x.M)))$ (de
\quad finition of $\Rightarrow$)
6. $\mathcal{C} \subseteq [F][C] \circ ([/-]M)$ (by assumption)
\quad $\subseteq [F][D] \circ (\text{map}(X.F)(y \cdot \text{coi}t_{X.F} (x.M)) \circ ([/-]M)$
\quad (monotonicity of $\circ$, previous line)
\quad $= [F][D] \circ (\text{map}(X.F)(y \cdot \text{coi}t_{X.F} (x.M))([/-]M))$ (previous
\quad line, $\mathcal{C}$ saturated)
\quad $\subseteq [F][D] \circ \text{out}(\text{coi}t_{X.F} (x.M) -))$ (by $\text{out}$ closure)
\quad $= [F][D] \circ \text{out} (\text{coi}t_{X.F} (x.M))$
7. $\mathcal{C} \subseteq [\nu X.F](\nu X.F) \circ (\text{coi}t_{X.F} (x.M))$ (adjunction $\star$
\quad)
8. $y : C \Rightarrow \text{coi}t_{X.F} (x.M) : [\nu X.F]$ (by definitions)

Corollary 34. If $x : C \vdash M : [\nu X.F]$ and $\Theta ; \Gamma \vdash N : C$ where $C$
\quad is saturated, then $\Theta ; \Gamma \vdash \text{coi}t_{X.F} (x.M) : N : [\nu X.F]$

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

Theorem 35 (Fundamental theorem). If $\Theta ; \Gamma \vdash M : A$, then we
\quad have $[\Theta ; \Gamma] : M \Rightarrow [A]$

Corollary 36. If $\Theta ; \Gamma \vdash M : A$ then for any $\alpha$, $M$ is strongly
\quad normalizing at $\alpha$.

7. Causality, Productivity and Liveness

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

We call a term $M$ a $\alpha$-value if it cannot step further at time $\alpha$.
\quad We may write this $M \not\Rightarrow_\alpha$. We obtain as a consequence of
\quad strong normalization that there is no closed inhabitant of the $\alpha$
\quad (which we define as $\mu X.X$), since there is no closed $\alpha$-value of
\quad this type. Perhaps surprisingly, we can also show there is no closed
\quad inhabitant of $\alpha \Rightarrow \bot$. For if there was some $\bot$ : $\alpha \Rightarrow \bot$
\quad we could evaluate $x : \bot ; \bot \Rightarrow f(x) : \bot$ at 0 to obtain a $\alpha$
\quad value $v : \bot ; v : \bot$, which by inspection of the possible $\alpha$-values,
\quad cannot exist!

Similarly, we can demonstrate an interesting property of the
\quad type $\nu X.X \cup X$. First, there is no closed term of type $\nu X.X \Rightarrow \bot$.
\quad For if there was some $f : \nu X.X \Rightarrow \bot$, we could normalize $x : B ; f
\quad \Rightarrow f(\text{in} \cdot x) : B$ at 0 to obtain a $\alpha$-value $x : B ; v : \bot$, which
\quad cannot exist. Moreover, there is no closed term of type $\nu X.X$, since
\quad there is no closed $\alpha$-value of type $\nu X.X$. That is, $\nu X.X$ is neither inhabi-
\quad ted nor provably uninhabited inside the logic.

To show how our operational semantics and strong normalization
\quad give rise to a causal (reactive) interpretation of programs, as
\quad well as an explanation of the liveness properties guaranteed by
\quad the types, we illustrate here the reactive execution of an example
\quad program $x_0 : \nu P \Rightarrow M_0 : \nu Q$. Such a program can be thought of as
\quad waiting for a $P$ event from its environment, and at some point deliv-
\quad ering a $Q$ event. For simplicity, we assume that $P$ and $Q$ are pure
\quad (non-temporal) types such as $\text{Bool}$ or $\mathbb{N}$. We consider sequences
\quad of interaction which begin as follows, where we write $\text{now } p$ for
\quad $\text{inj} (\text{in} p)$ and later $\bullet l$ for $\text{inj} (\text{in} l)$.
\quad
\begin{align*}
[\text{later } \bullet x_1/x_0]M_0 & \Rightarrow^* \text{later } \bullet M_1 \\
[\text{later } \bullet x_2/x_1]M_1 & \Rightarrow^* \text{later } \bullet M_2 \\
\ldots
\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
\quad step, and the program responding saying that it has not yet
\quad produced a $Q$ event. Essentially, at each stage, we leave a hole
\quad $x_i$ standing for input not yet known at this stage, which we will
\quad refine further in the next time step. The resulting $M_i$ acts as
\quad a continuation, specifying what to compute in the next time step.
\quad We note that each $x_i : \nu P \Rightarrow M_i : \nu Q$ by type preservation.
\quad Such a sequence may not end, if the environment defers providing a
\quad $P$ forever. However, it may end one of two ways. The first is if
\quad eventually the environment supplies a closed value $v$ of type $P$:
\quad
\begin{align*}
[\text{now } v/x_{i+1}]M_{i+1} & \Rightarrow^* v / \neq \omega
\end{align*}

In this case, $\vdash \nu \nu P/x_{i+1}]M_{i+1} : \nu Q$. By type preservation
\quad and strong normalization, we can evaluate this completely to $\vdash v
\quad / \neq \omega Q$. By inspection of the closed $\nu$-values, $v$ must be of the form
\quad $\nu v(x) (\nu (\ldots (\text{now } q)))$. That is, $\nu Q$ is eventually delivered in
\quad this case.

The second way such an interaction sequence may end is if the
\quad program produces a result before the environment has supplied a
\quad $P$:
\quad
\begin{align*}
[\text{later } \bullet x_{i+1}/x_i]M_i & \Rightarrow^* \text{now } q
\end{align*}

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

We also remark that strong normalization of $\Rightarrow^*$ guarantees
\quad a reactive productivity. That is, the evaluation of $[\text{later } \bullet x_{i+1}/x_i]M_i$
\quad is guaranteed to terminate at some $0$-value $v$ by strong normalization.
\quad As an aside, we note that if we were to use a time-indexed type
\quad system such as that of Krishnaswami and Benton [25], one could
\quad generalize this kind of argument to reduction at $n$, and normalize
\quad terms using $\Rightarrow^*_n$ to obtain $n$-value where $x$ can only occur at
time step $n+1$. This gives a more global perspective of reactive execu-
\quad tion. However, we use our form of the type system because we find
\quad it corresponds better in practice to how one thinks about reactive
\quad programs (one step at a time).

Finally, strong normalization of $\Rightarrow^*_\omega$ provides an explana-
\quad tion of liveness. When evaluating a closed term $\vdash M : \nu Q$ at $\omega$, we
\quad arrive at a closed $\omega$-value $v : \nu Q$ by inspection of the normal forms, such
\quad a value must provide a result after only finitely many delays. This is
to say that when running programs reactively, the environment
\quad may choose not to satisfy the requisite liveness requirement (e.g. it
\quad may never supply a $P$ event). In which case, the output of the program
\quad cannot reasonably be expected to guarantee its liveness property,
\quad since it may be waiting for an input event which never comes.
\quad However, we have the conditional result that if the environment
\quad 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 [24,25]. 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 [24,25], they obtain also promising results about space usage, which we have so far ignored in our foundation. In his most recent work, Krishnaswami [24] 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 $\nu X.X + \bigcirc X$. 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 [17,19] and Jeltsch [20], 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. His 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 [19], 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 Section 2 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 [20,21] 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 which enforce liveness properties in CPCs, unlike our work. He obtains the result that his Concrete Process Categories 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 observes that this triggers a collapse of (guarded) $\mu$ and $\nu$, so negative occurrences appear incompatible with our goals.

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 normalization, 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, it is commonly known that our iteration operators cannot express a constant-time predecessor function; one must traverse the entire term to compute its predecessor. This is straightforwardly solved by adding inverses to $\nu$ and out in the term language and operational semantics; the strong normalization proof extends easily. This is a motivation for studying the calculus with explicit $\mu$ and $\nu$ operators instead of merely using their impredicative encodings in a second order system such as (a hypothetical) System $F + \bigcirc$.

In a similar vein, the import functions we wrote in Section 2 unnecessarily traverse the entire term. From a foundational perspective, it is interesting to notice that they can be implemented at all. We remark that in our system, this can be done for types constructed without the negative connectives $\rightarrow$ and $\forall$. However, in practice, one would like to employ a device such as Krishnaswami’s stability [24] 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 – as Altenkirch and Danielsson [15] explain, Agda [32] has serious limitations with interleaving. Coq [7] does not fare any better. Some solutions exist employing sized types [1]. This is one motivation for using Nakano-style guarded recursion [16]. 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 $\nu TL$) – roughly, types built with only $\nu$, not $\mu$. This would be a step in the direction of unifying the two approaches. Similarly, it would be interesting to investigate pattern matching and especially the copatterns of Abel et al. [5] in this setting. This needs a careful treatment, because matching under $\wedge$ typically, but not always, violates causality.

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. [8] who study the presheaves on $\omega$, as well as Jeltsch [21] who studies more general (e.g. continuous) time domains. The idea is that the extra point “at infinity” expresses the global behaviour – it prevents (guarded) least and greatest fixed points from collapsing, and expresses our liveness guarantees, which can only be observed from a global perspective on time. This is essentially Jeltsch’s observation that, in his setting, only time domains having a greatest element allow one to express
liveness properties. The chief challenge 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**


