Fair Reactive Programming

Andrew Cave Francisco Ferreira Prakash Panangaden Brigitte Pientka

McGill University

{acave1,fferre8,prakash,bpientka}@cs.mcgill.ca

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 [17, 20, 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 intruiging 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 intepretation 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.

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, 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 • 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
     • xs' = tl xs
 in  ((hd fs) (hd xs)) :: (• (app fs' xs'))
```

We use the ◯ elimination form, **let** • 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 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 [24–26] – 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 [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  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 μ 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!

```
predictor : ◯ A → A
predictor x = let • x' = x in x'
     -- does not typecheck
```

Our typing rules disallow this program roughly by restricting variables bound under a • to only be usable under a •, in much the same way as Krishnaswami [24].

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:

```
predictor : ◯(A + B) → ◯ A + ◯ B
predictor x = let • x' = x in
 case x' of      -- does not typecheck
  | inl a ⇒ inl (• a)
  | inr a ⇒ inr (• 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, $\bigcirc\bot \to \bot$ 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:

```
import : A → ◯ A
import x = • x     -- 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 • when moving under a •. However, some concrete instances of this type are inhabited, for example the following program which brings natural numbers into the future:

```
import : Nat → ◯ Nat
import Zero = • Zero
import (Succ n) = let • n' = import n
                  in •(Succ n')
```

Our language does *not* have Nakano's guarded recursion [16] `fix : (◯ A → A) → 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 $\Diamond A$ in which the A is never delivered:

```
never : ◇ A
never = fix (λx. Later x)     -- 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.

```
repeat : A → □ A
repeat x = xs
  where xs = x :: •xs     -- 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.

```
data A 𝒰 B = Same of A × (◯ (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` $\mathcal{U}$ `B` is isomorphic to `(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` $\hat{\mathcal{U}}$ `B`.

```
type A 𝒰̂ B = A × ◯ (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:

```
codata Fair A B =
 In of (A 𝒰 (B 𝒰̂ (A × ◯(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`:

```
sched : □ A → □ B → Fair A B
sched as bs =
```

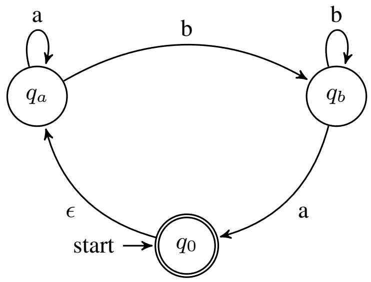
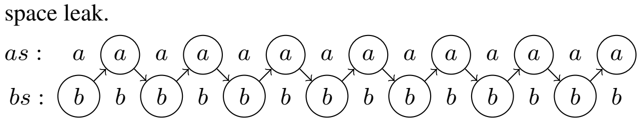


Figure 1. Scheduler automaton.

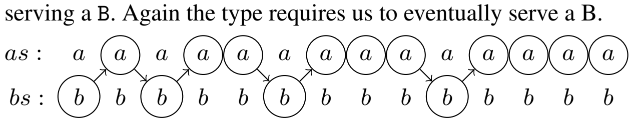
```
let • as' = tl as
    • bs' = tl bs
in
    In  (Switch (hd bs,
      • (Switch (hd as',
      let • as'' = tl as'
          • bs'' = tl bs'
      in • (sched as'' bs'')))))
```

The reader may notice that this scheduler drops the As at odd position and the Bs at even position. This could be overcome if one has `import` for the corresponding stream types, but at the cost of a space leak.



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 As each time before serving a B. Again the type requires us to eventually serve a 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

import n = case  n of
 | Zero ⇒ • Zero
 | Succ p ⇒ let  • n' = p in
    • (Succ (import n'))
```

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

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.

```
mutual
 cnt : TNat → TNat → □ A → □ B
  → (A 𝒰 (B Û (A × ◯ (Fair A B))))
 cnt n m as bs =
  let • m'  = import m
      • as' = tl as
      • bs' = tl bs
  in
  case  n of
  | Zero ⇒
   Switch (hd bs, • (Switch (hd as',
   let • m''  = import m'
       • as'' = tl as'
       • bs'' = tl bs'
   in
    • (sch2' (Succ (import m'')) as'' bs'')))
  | Succ p ⇒ let  • n' = p in
   Same (hd as, (• (cnt n' m' as' bs')))

and
  sch2' : TNat → □ A → □ B → Fair A B
  sch2' n as bs = In (cnt n n as bs)

−− Main function
sch2 : □ A → □ B → Fair A B
sch2 as bs = sch2' Zero as bs
```

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 (co)iteration 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 $\bigcirc A$ 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.

$$
\begin{array}{lcl}
\text{Types } A, B, F, G & ::= & 1 \mid F \times G \mid F + G \mid A \to F \mid \\
 & & \bigcirc F \mid \mu X.F \mid \nu X.F \mid X \\[1ex]
\text{Terms} \quad M, N & ::= & x \mid () \mid (M, N) \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\
\multicolumn{3}{l}{\quad \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N \mid \mathsf{inr}\ y \mapsto N'} \\
\multicolumn{3}{l}{\quad \mid \lambda x.M \mid M N \mid \bullet M \mid \mathsf{let}\ \bullet x = M\ \mathsf{in}\ N \mid \mathsf{inj}\ M} \\
\multicolumn{3}{l}{\quad \mid \mathsf{iter}_{X.F}\ (x.M)\ N \mid \mathsf{out}\ M \mid \mathsf{coit}_{X.F}\ (x.M)\ N} \\
\multicolumn{3}{l}{\quad \mid \mathsf{map}\ (\Delta.F)\ \eta\ M}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Contexts} & \Theta, \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Kind Contexts} & \Delta & ::= & \cdot \mid \Delta, X : * \\
\text{Type substitutions} & \rho & ::= & \cdot \mid \rho, F/X \\
\text{Morphisms} & \eta & ::= & \cdot \mid \eta, (x.M)/X
\end{array}
$$

Figure 3. LTL Syntax

Our term language is mostly standard and we only discuss the terms related to $\bigcirc$ modality and the fixed points, μ and ν. $\bullet M$ describes a term M which is available in the next time step. $\mathsf{let}\ \bullet x = M\ \mathsf{in}\ N$ 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 a $\bullet$. Our language also includes iteration operator $\mathsf{iter}_{X.F}\ (x.M)\ N$ and coiteration operator $\mathsf{coit}_{X.F}\ (x.M)\ N$. 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 μ types is $\mathsf{inj}\ M$, rolling up a term M. The elimination form for ν types is $\mathsf{out}\ M$, unrolling M.

We note that the $X.F$ annotations on iter and coit play a key role during runtime, since the operational semantics of map are defined using the structure of F (see Sec. 5). We do not annotate λ abstractions with their type because we are primarily interested in the operational behaviour of our language, and not e.g. unique typing. The term $\mathsf{map}\ (\Delta.F)\ \eta\ N$ 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.:

$$\begin{array}{rcl} \Box A & \equiv & \nu X.A \times \bigcirc X \\ \Diamond A & \equiv & \mu X.A + \bigcirc X \end{array}$$

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 great 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:

$$\mathsf{inf}\ A \quad \equiv \quad \nu X.\mu Y.(A \times \bigcirc X + \bigcirc Y)$$

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, $\Box\Diamond A$ 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, $\mathsf{inf} A$ can provide at most one value of type A at each time step. This demonstrates that the inclusion of general μ and ν 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 `Server`, $A\,\mathcal{U}\,B$, $A\,\hat{\mathcal{U}}\,B$ and $\texttt{Fair}\,A\,B$ which we used in the examples in Sec. 2.

$$\begin{array}{rcl} \texttt{Server} & \equiv & \nu X.\bigcirc X \times (\texttt{Req} \to \texttt{Resp} \times \bigcirc X) \\ A\,\mathcal{U}\,B & \equiv & \mu X.(B + A \times \bigcirc X) \\ A\,\hat{\mathcal{U}}\,B & \equiv & A \times \bigcirc(A\,\mathcal{U}\,B) \\ \texttt{Fair}\,A\,B & \equiv & \nu X.(A\,\mathcal{U}\,(B\,\hat{\mathcal{U}}\,(A \times \bigcirc X))) \end{array}$$

Finally, to illustrate the relationship between our formal language which features (co)iteration operators and the example programs which were written using general recusion, we show here the program *app* in our foundation. Here we need to use the pair of fs and xs as the coinduction invariant:

$$\begin{array}{l} app : \Box(A \to B) \to \Box A \to \Box B \\ app\ fs\ xs \equiv \mathsf{coit}_{X.B\times\bigcirc X}\ (x. \\ \quad \mathsf{let}\ \bullet\ fs' = tl\ (\mathsf{fst}\ x)\ \mathsf{in} \\ \quad \mathsf{let}\ \bullet\ xs' = tl\ (\mathsf{snd}\ x)\ \mathsf{in} \\ \qquad (\,(hd\ (\mathsf{fst}\ x))\ (hd\ (\mathsf{snd}\ x))\ ,\ \bullet(fs', xs')\,) \\ )\ (fs, xs) \end{array}$$

In the formal language, evapp becomes the following

$$\begin{array}{l} evapp : \Diamond A \to \Box(A \to B) \to \Diamond B \\ evapp \equiv \lambda ea.\ \mathsf{iter}^{\Box(A\to B)\to\Diamond B}_{X.A+\bigcirc X}\ (y. \\ \quad \mathsf{case}\ y\ \mathsf{of} \\ \qquad |\mathsf{inl}\ x \mapsto \lambda fs.\ \mathsf{inl}(\,(hd\ fs)\ x\,) \\ \qquad |\mathsf{inr}\ frec \mapsto \lambda fs. \\ \qquad\quad \mathsf{let}\ \bullet\ frec' = frec\ \mathsf{in} \\ \qquad\quad \mathsf{let}\ \bullet\ fs' = tl\ fs\ \mathsf{in} \\ \qquad\qquad \mathsf{inr}\,(\bullet\ (frec'\ fs')\,) \\ \quad)\ ea \end{array}$$

Well-formedness of types: $\Delta \vdash F : *$

$$\frac{}{\Delta \vdash 1 : *} \quad \frac{\Delta \vdash F : * \quad \Delta \vdash G : *}{\Delta \vdash F \times G : *} \quad \frac{\Delta \vdash F : * \quad \Delta \vdash G : *}{\Delta \vdash F + G : *}$$

$$\frac{\cdot \vdash A : * \quad \Delta \vdash F : *}{\Delta \vdash A \to F : *} \quad \frac{\Delta, X : * \vdash F : *}{\Delta \vdash \mu X.F : *} \quad \frac{\Delta, X : * \vdash F : *}{\Delta \vdash \nu X.F : *}$$

$$\frac{\Delta \vdash F : *}{\Delta \vdash \bigcirc F : *} \quad \frac{(X : *) \in \Delta}{\Delta \vdash X : *}$$

Figure 4. Well-formed types

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 $\to$. 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 Θ which provides types for variables which will become available in the next time step (i.e. when going under a $\bullet$) and the context Γ 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 Θ and Γ.

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 [25] or by using different judgments (i.e. 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 Γ 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 Γ.

The interesting rules are the ones for $\bigcirc$ modality and fixed points μ and ν. In the rule $I_{\bigcirc}$, the introduction rule for $\bigcirc$ 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 Θ move to the "available" position, while the assumptions in Γ are no longer available. In fact, Typing Rules for $\to, \times, +, 1$

$$\frac{\Theta; \Gamma, x{:}A \vdash M : B}{\Theta; \Gamma \vdash \lambda x.M : A \to B} \quad \frac{\Theta; \Gamma \vdash M : A \to B \quad \Theta; \Gamma \vdash N : A}{\Theta; \Gamma \vdash M N : B}$$

$$\frac{x{:}A \in \Gamma}{\Theta; \Gamma \vdash x : A} \quad \frac{}{\Theta; \Gamma \vdash () : 1} \quad \frac{\Theta; \Gamma \vdash M : A \quad \Theta; \Gamma \vdash N : B}{\Theta; \Gamma \vdash (M, N) : A \times B}$$

$$\frac{\Theta; \Gamma \vdash M : A \times B}{\Theta; \Gamma \vdash \mathsf{fst}\ M : A} \quad \frac{\Theta; \Gamma \vdash M : A \times B}{\Theta; \Gamma \vdash \mathsf{snd}\ M : B}$$

$$\frac{\Theta; \Gamma \vdash M : A}{\Theta; \Gamma \vdash \mathsf{inl}\ M : A + B} \quad \frac{\Theta; \Gamma \vdash N : B}{\Theta; \Gamma \vdash \mathsf{inr}\ N : A + B}$$

$$\frac{\Theta; \Gamma \vdash M : A + B \quad \Theta; \Gamma, x{:}A \vdash N_1 : C \quad \Theta; \Gamma, y{:}B \vdash N_2 : C}{\Theta; \Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : C}$$

Rules for $\bigcirc$ modality and least and greatest fixed points

$$\frac{\cdot; \Theta \vdash M : A}{\Theta; \Gamma \vdash \bullet M : \bigcirc A} \quad \frac{\Theta; \Gamma \vdash M : \bigcirc A \quad \Theta, x{:}A; \Gamma \vdash N : C}{\Theta; \Gamma \vdash \mathsf{let}\ \bullet x = M\ \mathsf{in}\ N : C}$$

$$\frac{\Theta; \Gamma \vdash M : [\mu X.F/X]F}{\Theta; \Gamma \vdash \mathsf{inj}\ M : \mu X.F} \quad \frac{\cdot; x{:}[C/X]F \vdash M : C \quad \Theta; \Gamma \vdash N : \mu X.F}{\Theta; \Gamma \vdash \mathsf{iter}_{X.F}\ (x.M)\ N : C}$$

$$\frac{\cdot; x{:}C \vdash M : [C/X]F \quad \Theta; \Gamma \vdash N : C}{\Theta; \Gamma \vdash \mathsf{coit}_{X.F}\ (x.M)\ N : \nu X.F} \quad \frac{\Theta; \Gamma \vdash M : \nu X.F}{\Theta; \Gamma \vdash \mathsf{out}\ M : [\nu X.F/X]F}$$

$$\frac{\Delta \vdash F : * \quad \Theta; \Gamma \vdash M : [\rho_1]F \quad \rho_1 \vdash \eta : \rho_2}{\Theta; \Gamma \vdash \mathsf{map}\ (\Delta.F)\ \eta\ M : [\rho_2]F}$$

Typing Rules for morphisms: $\rho_1 \vdash \eta : \rho_2$

$$\frac{\rho_1 \vdash \eta : \rho_2 \quad \cdot; x : A \vdash M : B}{\rho_1, A/X \vdash \eta, (x.M)/X : \rho_2, B/X} \quad \frac{}{\cdot \vdash \cdot : \cdot}$$

Figure 5. Typing Rules

the variables in Θ remain inaccessible until going under a $\bullet$ – this is how causality is enforced. Our motivation for forgetting about current assumptions Γ and not allowing them to persist is that we wish to obtain a type system which is sound for LTL. Allowing Γ to persist would make the unsound $A \to \bigcirc A$ easily derivable. In the work of Krishnaswami et. al [26], forgetting about current assumptions plays a key role in managing space usage.

The corresponding elimination form $\mathsf{let}\ \bullet x = M\ \mathsf{in}\ N$ simply adds $x{:}A$, the type of values promised by M, to the context Θ and we continue to check the body N allowing it to refer to x under a $\bullet$.

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

$$\frac{\cdot; x : 1 + C \vdash M : C \quad \Theta; \Gamma \vdash N : \mu X.1 + X}{\Theta; \Gamma \vdash \mathsf{iter}_{X.1+X}\ (x.M)\ N : C}$$

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 0 for the term $\mathsf{inj}\ (\mathsf{inl}())$ and $\mathsf{suc}\ m$ for the term $\mathsf{inj}\ (\mathsf{inr}\ m)$.

$$\begin{aligned} db &= \mathsf{iter}_{X.1+X}(x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \\ &\qquad\qquad\qquad\qquad \mid \mathsf{inr}\ w \mapsto \mathsf{suc}(\mathsf{suc}\ w)) \\ double &= \lambda x{:}\mathsf{Nat}.db\ x \end{aligned}$$

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

$$\begin{aligned} &unsound : A \to \Diamond B \to A\ \mathcal{U}\ B \\ &unsound \equiv \lambda a.\ \lambda eb.\ \mathsf{iter}_{X.B+\bigcirc X}\ (y. \\ &\quad \mathsf{case}\ y\ \mathsf{of} \\ &\qquad |\mathsf{inl}\ b \mapsto \mathsf{inj}\ (\mathsf{inl}\ b) \\ &\qquad |\mathsf{inr}\ u \mapsto \mathsf{inj}\ (\mathsf{inr}\ (a, u)) - a \text{ out of scope!} \\ &\quad)\ eb \end{aligned}$$

The typing rules for μ and ν 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:

$$\begin{tikzcd} F(\mu F) \arrow[r, dashed, "F(\mathsf{iter} f)"] \arrow[d, "\mathsf{inj}"'] & F(C) \arrow[d, "f"] \\ \mu F \arrow[r, dashed, "\mathsf{iter}\ f"] & C \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, dashed, "\mathsf{coit} f"] \arrow[d, "f"'] & \nu F \arrow[d, "\mathsf{out}"] \\ F(C) \arrow[r, dashed, "F(\mathsf{coit} f)"] & F(\nu F) \end{tikzcd}$$

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.

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

The term $\mathsf{map}\ (\Delta.F)\ \eta\ M$ witnesses that F is a *functor*. It has the effect of applying the transformations specified by η 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, Δ binds the free variables of F. It is illustrative to consider the case where $F(Y) = \mathsf{list}\ Y = \mu X.1 + Y \times X$. If $y : A \vdash M : B$ and $N : \mathsf{list}\ A$, then $\mathsf{map}\ (Y.\ \mathsf{list}\ Y)\ ((y.\ M)/Y)\ N : \mathsf{list}\ B$. 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 $[N/x]^\bullet M$. The key case in their definition is the case for $\bullet M$. For current substitution $[N/x](\bullet M)$, in well-scoped terms, x cannot occur in M, so we define:

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

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

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

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* $\Theta; \Gamma \vdash [N/x]M : B$

$$
\begin{array}{rcl}
\mathcal{E}_{k+1} & ::= & \bullet\mathcal{E}_k \\
\mathcal{E}_k & ::= & \mathcal{E}_k\ N \mid M\ \mathcal{E}_k \mid \mathsf{fst}\ \mathcal{E}_k \mid \mathsf{snd}\ \mathcal{E}_k \mid (\mathcal{E}_k, M) \mid (M, \mathcal{E}_k) \\
& & \mid \lambda x.\mathcal{E}_k \mid \mathsf{case}\ \mathcal{E}_k\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N \mid \mathsf{inr}\ y \mapsto N' \\
& & \mid \mathsf{inl}\ \mathcal{E}_k \mid \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto \mathcal{E}_k \mid \mathsf{inr}\ y \mapsto N \\
& & \mid \mathsf{inr}\ \mathcal{E}_k \mid \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N \mid \mathsf{inr}\ y \mapsto \mathcal{E}_k \\
& & \mid \mathsf{let}\ \bullet x = \mathcal{E}_k\ \mathsf{in}\ N \mid \mathsf{let}\ \bullet x = M\ \mathsf{in}\ \mathcal{E}_k \mid \mathsf{inj}\ \mathcal{E}_k \\
& & \mid \mathsf{iter}_{X.F}\ (x.M)\ \mathcal{E}_k \ \mid \mathsf{coit}_{X.F}\ (x.M)\ \mathcal{E}_k \\
& & \mid \mathsf{out}\ \mathcal{E}_k \mid \mathsf{map}\ (\Delta.F)\ \eta\ \mathcal{E}_k \\
\mathcal{E}_0 & ::= & []
\end{array}
$$

Figure 6. Evaluation contexts

$$
\begin{array}{lll}
(\lambda x.M)\ N & \longrightarrow & [N/x]M \\
\mathsf{fst}(M, N) & \longrightarrow & M \\
\mathsf{snd}(M, N) & \longrightarrow & N \\
\mathsf{case}\ (\mathsf{inl}\ M)\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 & \longrightarrow & [M/x]N_1 \\
\mathsf{case}\ (\mathsf{inr}\ M)\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 & \longrightarrow & [M/y]N_2 \\
\mathsf{let}\ \bullet x = \bullet M\ \mathsf{in}\ N & \longrightarrow & [M/x]^{\bullet}N \\
\mathsf{iter}_{X.F}\ (x.M)\ (\mathsf{inj}\ N) & \longrightarrow & \\
\quad [\mathsf{map}\ (X.F)\ ((y.\ \mathsf{iter}_{X.F}\ (x.M)\ y)/X)\ N/x]M & & \\
\mathsf{out}\ (\mathsf{coit}_{X.F}\ (x.M)\ N) & \longrightarrow & \\
\quad \mathsf{map}\ (X.F)\ ((y.\ \mathsf{coit}_{X.F}\ (x.M)\ y)/X)\ ([N/x]M) & &
\end{array}
$$

Figure 7. Operational Semantics

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

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 $\bigcirc$ modality. We define evaluation contexts in Fig. 6. We index an evaluation context $\mathcal{E}_k$ by a depth k which indicates how many $\bigcirc$ 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 $\bigcirc$ 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 $\mathcal{E}_k[M]$ to $\mathcal{E}_k[N]$ 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:

$$
\frac{M \longrightarrow N}{\mathcal{E}_k[M] \rightsquigarrow_\alpha \mathcal{E}_k[N]}\ \text{if } k \leq \alpha
$$

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 α 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 $(\mathsf{out}\ (\mathsf{coit}_{X.F}\ (x.M)\ N))$, 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 $(\mathsf{iter}_{X.F}\ (x.M)\ (\mathsf{inj}\ N))$, 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 μ and ν. For μ, we reduce N until it is of the form inj N. At which point, we can continue applying η inside N, where now at the recursive positions specified by Y we need to continue recursively applying map. The case for ν 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 iter, coit, and 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 explictly 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 $double$ which multiplies a given natural number by two given in the previous section. Recall the following abbreviations for natural numbers: $0 = \mathsf{inj}\ (\mathsf{inl}())$, $\mathsf{suc}\ w = \mathsf{inj}\ (\mathsf{inr}\ w)$, $1 = \mathsf{suc}\ 0$, etc.

Let us first compute $double\ 0$.

$$
\begin{array}{l}
\quad double\ 0 \\
\longrightarrow db\ (\mathsf{inj}\ (\mathsf{inl}\ ())) \\
\longrightarrow \mathsf{case}\ M_0\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \mid \mathsf{inr}\ w \mapsto \mathsf{suc}\ (\mathsf{suc}\ w)
\end{array}
$$

where

$$
\begin{array}{rl}
M_0 & = \mathsf{map}\ (1 + X)\ (y.db\ y/X)\ (\mathsf{inl}\ ()) \\
& \longrightarrow^* \mathsf{case}\ (\mathsf{inl}())\ \mathsf{of}\ \mathsf{inl}\ v \mapsto \mathsf{inl}() \mid \mathsf{inr}\ u \mapsto \mathsf{inr}(db\ u)
\end{array}
$$

$$
\begin{array}{l}
\longrightarrow^* \mathsf{case}\ (\mathsf{inl}\ ())\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \mid \mathsf{inr}\ w \mapsto \mathsf{suc}\ (\mathsf{suc}\ w) \\
\longrightarrow\ 0
\end{array}
$$

We now compute $double\ 1$

$$
\begin{array}{l}
\quad double\ 1 \\
\longrightarrow db\ (\mathsf{inj}\ (\mathsf{inr}\ 0)) \\
\longrightarrow \mathsf{case}\ M_1\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \mid \mathsf{inr}\ w \mapsto \mathsf{suc}\ (\mathsf{suc}\ w)
\end{array}
$$

where

$$
\begin{array}{rl}
M_1 & = \mathsf{map}\ (1 + X)\ (y.db\ y/X)\ (\mathsf{inr}\ 0) \\
& \longrightarrow^* \mathsf{case}\ (\mathsf{inr}\ 0)\ \mathsf{of}\ \mathsf{inl}\ v \mapsto \mathsf{inl}() \mid \mathsf{inr}\ u \mapsto \mathsf{inr}(db\ u)
\end{array}
$$

$$
\begin{array}{l}
\longrightarrow^* \mathsf{case}\ (\mathsf{inr}(db\ 0))\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \mid \mathsf{inr}\ w \mapsto \mathsf{suc}\ (\mathsf{suc}\ w) \\
\longrightarrow^* \mathsf{case}\ (\mathsf{inr}\ 0)\ \mathsf{of}\ \mathsf{inl}\ y \mapsto 0 \mid \mathsf{inr}\ w \mapsto \mathsf{suc}\ (\mathsf{suc}\ w) \\
\longrightarrow\ \mathsf{suc}\ (\mathsf{suc}\ 0) = 2
\end{array}
$$

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

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

Observe that after evaluating $M \rightsquigarrow_n^* N$, where N is in normal form, one can then resume evaluation $N \rightsquigarrow_{n+1}^* N'$ 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 $\rightsquigarrow_0$ (see below), however considering the general $\rightsquigarrow_\alpha$ gives us the tools to analyze programs from a more global viewpoint, which is important for liveness guarantees.

$$
\begin{array}{llll}
\mathsf{map} & (\Delta.\ X)\ \eta & N \longrightarrow [N/x]M & \text{if } \eta(X) = x.M \\
\mathsf{map} & (\Delta.\ 1)\ \eta & N \longrightarrow () & \\
\mathsf{map} & (\Delta.\ F \times G)\ \eta & N \longrightarrow (\mathsf{map}\ (\Delta.\ F)\ \eta\ (\mathsf{fst}\ N), \mathsf{map}\ (\Delta.\ G)\ \eta\ (\mathsf{snd}\ N)) & \\
\mathsf{map} & (\Delta.\ F + G)\ \eta & N \longrightarrow \mathsf{case}\ N\ \mathsf{of}\ \mathsf{inl}\ y \mapsto \mathsf{inl}(\mathsf{map}\ (\Delta.\ F)\ \eta\ y) & \\
 & & \qquad\qquad\qquad\quad \mid \mathsf{inr}\ z \mapsto \mathsf{inr}(\mathsf{map}\ (\Delta.\ G)\ \eta\ z) & \\
\mathsf{map} & (\Delta.\ A \to F)\ \eta & N \longrightarrow \lambda y.\mathsf{map}\ (\Delta.\ F)\ \eta\ (N\ y) & \\
\mathsf{map} & (\Delta.\ \bigcirc F)\ \eta & N \longrightarrow \mathsf{let}\ \bullet y = N\ \mathsf{in}\ \bullet\,\mathsf{map}\ (\Delta.\ F)\ \eta\ y & \\
\mathsf{map} & (\Delta.\ \mu Y.F)\ \eta\ (\mathsf{inj}\ N) & \longrightarrow \mathsf{inj}\ (\mathsf{map}\ (\Delta, Y.\ F)\ (\eta, (x.\ \mathsf{map}\ (\Delta.\ \mu Y.F)\ \eta\ x)/Y)\ N) & \\
\mathsf{out}\ (\mathsf{map} & (\Delta.\ \nu Y.F)\ \eta & N) \longrightarrow \mathsf{map}\ (\Delta, Y.\ F)\ (\eta, (x.\ \mathsf{map}\ (\Delta.\ \nu Y.F)\ \eta\ x)/Y)\ (\mathsf{out}\ N) &
\end{array}
$$

Figure 8. Operational semantics of map

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

Proposition 3.

1. *If $N \rightsquigarrow_\alpha N'$ then $[N/x]M \rightsquigarrow_\alpha^* [N'/x]M$*
2. *If $M \rightsquigarrow_\alpha M'$ then $[N/x]M \rightsquigarrow_\alpha [N/x]M'$*
3. *If $N \rightsquigarrow_n N'$ then $[N/x]^\bullet M \rightsquigarrow_{n+1}^* [N'/x]^\bullet M$*
4. *If $N \rightsquigarrow_\omega N'$ then $[N/x]^\bullet M \rightsquigarrow_\omega^* [N'/x]^\bullet M$*
5. *If $M \rightsquigarrow_\alpha M'$ then $[N/x]^\bullet M \rightsquigarrow_\alpha [N/x]^\bullet M'$*

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 $M \rightsquigarrow_\alpha ...$ is finite, for well-typed terms M. This means that one can compute the approximate value of M up to time α. In fact, this result actually gives us more: our programs are suitably causal using normalization at 0, and the type of a program gives rise to a liveness property which it satisfies, using normalization at ω. 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 $\omega + 1$ is discrete, i.e. $\alpha \leq \beta$ precisely when $\alpha = \beta$. It is not step-indexed because our strict positivity condition on μ and ν means we can interpret them without the aid of step indexing.

This proof is made challenging by the presence of interleaving μ and ν types and (co)iteration operators. This is uncommon, but 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 operation, which we find simplifies the proof substantially as it allows us to consider map independently of iter and coit in the proof.

We use a standard inductive characterization of strongly normalizing:

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

$$\frac{\forall M', M \rightsquigarrow_\alpha M' \Longrightarrow M' \in sn_\alpha}{M \in sn_\alpha}$$

We say M is strongly normalizing at α *if $M \in sn_\alpha$.*

It is immediate from this definition that if $M \in sn_\alpha$ and $M \rightsquigarrow_\alpha M'$ then $M' \in sn_\alpha$. Since this is an inductive definition, it affords us a corresponding induction principle: To show that a property P holds of a term $M \in sn_\alpha$, one is allowed to assume that P holds for all M' such that $M \rightsquigarrow_\alpha M'$. One can easily verify by induction that if $M \in sn_\alpha$ then there are no infinite $\rightsquigarrow_\alpha$ 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 [28], 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. $\mathcal{A} : \omega + 1 \to \mathcal{P}(tm)$, where we write tm for the set of all terms. We overload the notation $\subseteq$, $\cap$, and $\cup$ to mean pointwise inclusion, intersection, and union. That is, if $\mathcal{A}$ and $\mathcal{B}$ are indexed sets, we will write $\mathcal{A} \subseteq \mathcal{B}$ to mean $\mathcal{A}^\alpha \subseteq \mathcal{B}^\alpha$ for all α.

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

$$
\begin{array}{lcl}
(\blacktriangleright \mathcal{A})^0 & \equiv & tm \\
(\blacktriangleright \mathcal{A})^{m+1} & \equiv & \mathcal{A}^m \\
(\blacktriangleright \mathcal{A})^\omega & \equiv & \mathcal{A}^\omega
\end{array}
$$

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 $M \twoheadrightarrow_\alpha M'$ for a contraction, and $M \rightharpoonup_\alpha N$ for a contraction occuring 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.

$$
\begin{array}{lcl}
\mathcal{H} & ::= & [\,] \mid \mathsf{fst}\ \mathcal{H} \mid \mathsf{snd}\ \mathcal{H} \mid \mathcal{H} N \mid \mathsf{iter}_{X.F}\ (x.M)\ \mathcal{H} \\
 & & \mid \mathsf{map}\ (\Delta.\ \mu X.F)\ \eta\ \mathcal{H} \mid \mathsf{out}\ \mathcal{H} \mid \mathsf{let}\ \bullet x = \mathcal{H}\ \mathsf{in}\ N \\
 & & \mid \mathsf{case}\ \mathcal{H}\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2
\end{array}
$$

$$\frac{M \twoheadrightarrow_\alpha M'}{\mathcal{H}[M] \rightharpoonup_\alpha \mathcal{H}[M']}$$

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

$$\frac{N \in sn_\alpha}{(\lambda x.M)\; N \twoheadrightarrow_\alpha [N/x]M} \qquad \frac{N \in sn_\alpha}{\mathsf{fst}(M,N) \twoheadrightarrow_\alpha M} \qquad \frac{M \in sn_\alpha}{\mathsf{snd}(M,N) \twoheadrightarrow_\alpha N} \qquad \frac{N \in (\blacktriangleright sn)_\alpha}{(\mathsf{let}\ \bullet x = \bullet N \ \mathsf{in}\ M) \twoheadrightarrow_\alpha [N/x]^\bullet M}$$

$$\frac{M \in sn_\alpha \quad N_2 \in sn_\alpha}{\mathsf{case}\,(\mathsf{inl}\,M)\,\mathsf{of}\ \mathsf{inl}\,x \mapsto N_1 \mid \mathsf{inr}\,y \mapsto N_2 \twoheadrightarrow_\alpha [M/x]N_1} \qquad \frac{M \in sn_\alpha \quad N_1 \in sn_\alpha}{\mathsf{case}\,(\mathsf{inr}\,M)\,\mathsf{of}\ \mathsf{inl}\,x \mapsto N_1 \mid \mathsf{inr}\,y \mapsto N_2 \twoheadrightarrow_\alpha [M/y]N_2}$$

$$\frac{N \in sn_\alpha}{\mathsf{iter}_{X.F}\,(x.M)\,(\mathsf{inj}\ N) \ \twoheadrightarrow_\alpha [\mathsf{map}\ \ F\ \ (y.\ \mathsf{iter}_{X.F}\,(x.M)\,y\ /X)\ \ N\ /x]M}$$

$$\frac{N \in sn_\alpha}{(\mathsf{out}\ \ (\mathsf{coit}_{X.F}\,(x.M)\,N)) \twoheadrightarrow_\alpha \mathsf{map}\,F\,(y.\ \mathsf{coit}_{X.F}\,(x.M)\,y)\,([N/x]M)}$$

$$\frac{\eta(X) = x.M \quad N \in sn_\alpha}{\mathsf{map}\ (\Delta.\ X)\ \eta\ N \twoheadrightarrow_\alpha [N/x]M} \qquad \frac{N \in sn_\alpha}{\mathsf{map}\ (\Delta.\ 1)\ \eta\ N \twoheadrightarrow_\alpha ()}$$

$$\frac{}{\mathsf{map}\ (\Delta.\ F \times G)\ \eta\ N \twoheadrightarrow_\alpha (\mathsf{map}\ (\Delta.\ F)\ \eta\ (\mathsf{fst}\ N), \mathsf{map}\ (\Delta.\ G)\ \eta\ (\mathsf{snd}\ N))}$$

$$\frac{}{\mathsf{map}\ (\Delta.\ F + G)\ \eta\ N \twoheadrightarrow_\alpha \mathsf{case}\,N\,\mathsf{of}\ \mathsf{inl}\,y \mapsto \mathsf{inl}(\mathsf{map}\ (\Delta.\ F)\ \eta\ y) \mid \mathsf{inr}\,z \mapsto \mathsf{inr}(\mathsf{map}\ (\Delta.\ G)\ \eta\ z)}$$

$$\frac{}{\mathsf{map}\ (\Delta.\ A \to F)\ \eta\ N \twoheadrightarrow_\alpha \lambda y.\mathsf{map}\ (\Delta.\ F)\ \eta\ (N\,y)}$$

$$\frac{}{\mathsf{map}\ (\Delta.\ \bigcirc F)\ \eta\ N \twoheadrightarrow_\alpha \mathsf{let}\ \bullet y = N\ \mathsf{in}\ \bullet(\mathsf{map}\ (\Delta.\ F)\ \eta\ y)}$$

$$\frac{}{\mathsf{map}\ (\Delta.\ \mu Y.F)\ \eta\ (\mathsf{inj}\ N) \twoheadrightarrow_\alpha \mathsf{inj}\ (\mathsf{map}\ (\Delta, Y.\ F)\ (\eta, (x.\ \mathsf{map}\ (\Delta.\ \mu Y.F)\ \eta\ x)/Y)\ N)}$$

$$\frac{}{\mathsf{out}\ (\mathsf{map}\ (\Delta.\ \nu Y.F)\ \eta\ N) \twoheadrightarrow_\alpha \mathsf{map}\ (\Delta, Y.\ F)\ (\eta, (x.\ \mathsf{map}\ (\Delta.\ \nu Y.F)\ \eta\ x)/Y)\ (\mathsf{out}\ N)}$$

Figure 9. Normalizing weak head reduction

Proof. We consider each redex in the definition of $\twoheadrightarrow$. The proofs are by lexicographic induction on the derivations of sn_α. We show some of the cases of interest:

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

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

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

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

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

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

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

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

We analyze the ways $\mathcal{H}[\mathsf{let}\ \bullet x = \bullet N\ \mathsf{in}\ M]$ can reduce:

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

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

(c) $\bullet N \rightsquigarrow_\alpha \bullet N'$:

The case $\alpha = 0$ is impossible.

If $\alpha = m + 1$ then $N \rightsquigarrow_m N'$ and $\mathcal{H}[[N/x]^\bullet M] \rightsquigarrow_{m+1}^* \mathcal{H}[[N'/x]^\bullet M]$, and hence $\mathcal{H}[[N'/x]^\bullet M] \in sn_{m+1}$. By the induction hypothesis with the smaller derivation $N' \in sn_m$, we have $\mathcal{H}[\mathsf{let}\ \bullet x = \bullet N'\ \mathsf{in}\ M] \in sn_{m+1}$ as required.

If $\alpha = \omega$ then $N \rightsquigarrow_\omega N'$ and $\mathcal{H}[[N/x]^\bullet M] \rightsquigarrow_\omega^* \mathcal{H}[[N'/x]^\bullet M]$, and hence $\mathcal{H}[[N'/x]^\bullet M] \in sn_\omega$. By the induction hypothesis with the smaller derivation $N' \in sn_\omega$, we have $\mathcal{H}[\mathsf{let}\ \bullet x = \bullet N'\ \mathsf{in}\ M] \in sn_\omega$ as required.

(d) $\mathsf{let}\ \bullet x = \bullet N\ \mathsf{in}\ M \rightsquigarrow_\alpha [N/x]^\bullet M$: we have $\mathcal{H}[[N/x]^\bullet M] \in sn_\alpha$ by assumption.

The other cases are similar. □

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

We define the indexed set of terms sne (“strongly normalizing neutral”) which are strongly normalizing, but are stuck with a variable in place of a weak head redex.

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

Lemma 8. *The rules presented in Figure 10 are admissible for* sn *and* sne*.*

Proof. We show the case $\blacktriangleright sn \subseteq sn \circ \bullet$:

Suppose $M \in (\blacktriangleright sn)_\alpha$.

Case $\alpha = 0$: then $\bullet M \in sn_0$ because there are no M' such that $\bullet M \rightsquigarrow_0 M'$.

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

Suppose $\bullet M \rightsquigarrow_{m+1} N$. By inversion, we have $N = \bullet M'$ and $M \rightsquigarrow_m M'$. By I.H. $\bullet M' \in sn_{m+1}$ as required.

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

Suppose $\bullet M \rightsquigarrow_\omega N$. By inversion, we have $N = \bullet M'$ and $M \rightsquigarrow_\omega M'$. 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 $\mathcal{A} : \omega + 1 \to \mathcal{P}(tm)$ is* saturated *if*

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

It is immediate from Lemma 6 that sn is saturated.

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

$$\overline{\mathcal{A}}^\alpha \equiv \{M | \exists M'. M \twoheadrightarrow^*_\alpha M' \land (M' \in \mathcal{A}^\alpha \lor M' \in sne_\alpha)\}$$

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

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

$$\bigwedge \mathcal{S} \equiv (\bigcap \mathcal{S}) \cap sn \qquad \bigvee \mathcal{S} \equiv \overline{\bigcup \mathcal{S}}$$

For $\land$, we intersect with sn so that the nullary lower bound is sn, and hence saturated. For non-empty $\mathcal{S}$, we have $\bigwedge \mathcal{S} = \bigcap \mathcal{S}$. 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:*

$$\begin{array}{lcl} \mu F & \equiv & \bigwedge\{\mathcal{C} \text{ saturated} \mid F(\mathcal{C}) \subseteq \mathcal{C}\} \\ \nu F & \equiv & \bigvee\{\mathcal{C} \text{ saturated} \mid \mathcal{C} \subseteq F(\mathcal{C})\} \end{array}$$

If F is monotone and takes saturated sets to saturated sets, then μF (resp. νF) 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 $\mathcal{C}$ is saturated, then:*

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

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

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

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

□

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

Definition 16. *We define:*

$$\begin{array}{lcl} (\mathcal{A} \circ f)^\alpha & \equiv & \{M | fM \in \mathcal{A}^\alpha\} \\ (\mathcal{A} \star f)^\alpha & \equiv & \{fM | M \in \mathcal{A}^\alpha\} \end{array}$$

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

Lemma 17. *The following properties hold:*

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

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 $\mathcal{H}[M] \in sn_\alpha$. Suppose $M \rightsquigarrow_\alpha M'$. Then $\mathcal{H}[M] \rightsquigarrow_\alpha \mathcal{H}[M']$. Hence by the induction hypothesis, $M' \in sn_\alpha$ as required. □

Lemma 19. *If $\mathcal{A}$ is saturated and $\mathcal{H}$ is a head context then $\mathcal{A} \circ \mathcal{H}$ is saturated.*

Proof.
1. $\mathcal{A} \circ \mathcal{H} \subseteq sn \circ \mathcal{H} \subseteq sn$ (by lemmas 18,17)
2. If $\mathcal{H}'[M] \twoheadrightarrow_\alpha \mathcal{H}'[M']$ and $\mathcal{H}'[M'] \in (\mathcal{A} \circ \mathcal{H})^\alpha$ then $\mathcal{H}[\mathcal{H}'[M]] \twoheadrightarrow_\alpha \mathcal{H}[\mathcal{H}'[M']]$ and $\mathcal{H}[\mathcal{H}'[M']] \in \mathcal{A}^\alpha$, hence $\mathcal{H}[\mathcal{H}'[M]] \in \mathcal{A}^\alpha$ (by closedness of $\mathcal{A}$).
3. $sne \subseteq sne \circ \mathcal{H} \subseteq \mathcal{A} \circ \mathcal{H}$ (by lemmas 8,17) □

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:*

$$\begin{array}{lcl} \mathbf{1} & \equiv & sn \\ (\mathcal{A} \times \mathcal{B}) & \equiv & \overline{(\mathcal{A} \circ \mathsf{fst}) \cap (\mathcal{B} \circ \mathsf{snd})} \\ (\mathcal{A} + \mathcal{B}) & \equiv & \overline{(\mathcal{A} \star \mathsf{inl}\,) \cup (\mathcal{B} \star \mathsf{inr}\,)} \\ (\mathcal{A} \to \mathcal{B})^\alpha & \equiv & \overline{\{M | \forall N \in \mathcal{A}^\alpha. (MN) \in \mathcal{B}^\alpha\}} \\ (\bigcirc \mathcal{A}) & \equiv & \overline{(\blacktriangleright \mathcal{A}) \star \bullet} \\ \boldsymbol{\mu}\mathcal{F} & \equiv & \mu(\mathcal{X} \mapsto \overline{\mathcal{F}(\mathcal{X}) \star \mathsf{inj}}) \\ \boldsymbol{\nu}\mathcal{F} & \equiv & \nu(\mathcal{X} \mapsto \mathcal{F}(\mathcal{X}) \circ \mathsf{out}) \end{array}$$

Notice that $\boldsymbol{\mu}\mathcal{F}$ is defined regardless of whether $\mathcal{F}$ is monotone, although we only know that $\boldsymbol{\mu}\mathcal{F}$ is actually a least fixed point when $\mathcal{F}$ is monotone. We remark also that our $\to$ 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,

$$\frac{}{x \in sne_\alpha} \qquad \frac{M \in sne_\alpha \quad N \in sn_\alpha}{M\ N \in sne_\alpha} \qquad \frac{M \in sne_\alpha}{\mathsf{fst}\ M \in sne_\alpha} \qquad \frac{M \in sne_\alpha}{\mathsf{snd}\ M \in sne_\alpha} \qquad \frac{M \in sne_\alpha}{\mathsf{out}\ M \in sne_\alpha} \qquad \frac{N \in sne_\alpha}{\mathsf{map}\ (\Delta.\ \mu X.F)\ \eta\ N \in sne_\alpha}$$

$$\frac{M \in sne_\alpha \quad N_1 \in sn_\alpha \quad N_2 \in sn_\alpha}{\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 \in sne_\alpha} \qquad \frac{N \in sne_\alpha}{\mathsf{iter}_{X.F}\ (x.M)\ \ N \in sne_\alpha} \qquad \frac{M \in sne_\alpha \quad N \in sn_\alpha}{\mathsf{let}\ \bullet x = M\ \mathsf{in}\ N \in sne_\alpha}$$

$$\frac{M \in sne_\alpha}{M \in sn_\alpha} \qquad \frac{M \in sn_\alpha}{\lambda x.M \in sn_\alpha} \qquad \frac{M \in sn_\alpha \quad N \in sn_\alpha}{(M, N) \in sn_\alpha} \qquad \frac{M \in sn_\alpha}{\mathsf{inl}\ M \in sn_\alpha} \qquad \frac{M \in sn_\alpha}{\mathsf{inr}\ M \in sn_\alpha} \qquad \frac{M \in sn_\alpha}{\mathsf{inj}\ M \in sn_\alpha}$$

$$\frac{N \in sn_\alpha}{\mathsf{coit}_{X.F}\ (x.M)\ N \in sn_\alpha} \qquad \frac{M \in (\blacktriangleright sn)_\alpha}{\bullet M \in sn_\alpha} \qquad \frac{N \in sn_\alpha}{\mathsf{map}\ (\Delta.\ \nu X.F)\ \eta\ N \in sn_\alpha}$$

Figure 10. Admissible rules for sn and sne

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 $\mathbf{1}, \times, \boldsymbol{\mu}, \boldsymbol{\nu}$ observe that previous lemmas establish that the operations involved in their definitions take saturated sets to saturated sets.

Case $\mathcal{A} + \mathcal{B}$: Suppose $\mathcal{A}$ and $\mathcal{B}$ are saturated.

1. $\mathcal{A} \subseteq sn$ (since $\mathcal{A}$ saturated)
   $\subseteq sn \circ \mathsf{inl}$ (by lemma 8)
2. $\mathcal{A} \star \mathsf{inl} \subseteq sn$ (by lemma 17)
3. $\mathcal{B} \star \mathsf{inr} \subseteq sn$ (similarly)
4. $(\mathcal{A} \star \mathsf{inl}) \cup (\mathcal{B} \star \mathsf{inr}) \subseteq sn$
5. $\overline{(\mathcal{A} \star \mathsf{inl}) \cup (\mathcal{B} \star \mathsf{inr})}$ is saturated (by lemma 11)

Case $\bigcirc\mathcal{A}$: Suppose $\mathcal{A}$ is saturated.

1. $\mathcal{A} \subseteq sn$ (since $\mathcal{A}$ saturated)
2. $\blacktriangleright \mathcal{A} \subseteq \blacktriangleright sn$ (functor)
   $\subseteq sn \circ \bullet$ (by lemma 8)
3. $\blacktriangleright \mathcal{A} \star \bullet \subseteq sn$ (by lemma 17)
4. $\overline{\blacktriangleright \mathcal{A} \star \bullet}$ is saturated (by lemma 11)

Case $\mathcal{A} \to \mathcal{B}$: Suppose $\mathcal{A}$ and $\mathcal{B}$ are saturated.

1. Suppose $M \in (\mathcal{A} \to \mathcal{B})^\alpha$, i.e. for any $N \in \mathcal{A}^\alpha$ we have $M\ N \in \mathcal{B}^\alpha$.
   (a) $x \in \mathcal{A}^\alpha$ (since $\mathcal{A}$ is saturated)
   (b) $M\ x \in \mathcal{B}^\alpha$ (by assumption)
   (c) $M\ x \in sn_\alpha$ (since $\mathcal{B}$ is saturated)
   (d) $M \in sn_\alpha$ (by lemma 18)
   So $(\mathcal{A} \to \mathcal{B}) \subseteq sn$
2. Suppose $M \rightharpoonup_\alpha M'$ with $M' \in (\mathcal{A} \to \mathcal{B})^\alpha$, i.e. for any $N \in \mathcal{A}^\alpha$ we have $M'\ N \in \mathcal{B}^\alpha$. Suppose $N \in \mathcal{A}^\alpha$.
   (a) $MN \rightharpoonup_\alpha M'N$ (by def of $\rightharpoonup$)
   (b) $MN \in \mathcal{B}^\alpha$ (since $\mathcal{B}$ is saturated)
   So $\mathcal{A} \to \mathcal{B}$ is backward closed under $\rightharpoonup$.
3. Suppose $M \in sne_\alpha$. Suppose $N \in \mathcal{A}^\alpha$. Then $MN \in sne_\alpha$, hence $MN \in \mathcal{B}^\alpha$ (since $\mathcal{B}$ is saturated). Hence $M \in (\mathcal{A} \to \mathcal{B})$
   Hence $sne \subseteq \mathcal{A} \to \mathcal{B}$

□

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

Definition 22. *Given ρ, 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:*

$$\begin{array}{lcl}
[\![X]\!](\rho) & \equiv & \rho(X) \\
[\![1]\!](\rho) & \equiv & \mathbf{1} \\
[\![F \times G]\!](\rho) & \equiv & [\![F]\!](\rho) \times [\![G]\!](\rho) \\
[\![F + G]\!](\rho) & \equiv & [\![F]\!](\rho) + [\![G]\!](\rho) \\
[\![A \to F]\!](\rho) & \equiv & [\![A]\!](\cdot) \to [\![F]\!](\rho) \\
[\![\bigcirc F]\!](\rho) & \equiv & \bigcirc[\![F]\!](\rho) \\
[\![\mu X.F]\!](\rho) & \equiv & \boldsymbol{\mu}(\mathcal{X} \mapsto [\![F]\!](\rho, \mathcal{X}/X)) \\
[\![\nu X.F]\!](\rho) & \equiv & \boldsymbol{\nu}(\mathcal{X} \mapsto [\![F]\!](\rho, \mathcal{X}/X))
\end{array}$$

Observe that, by lemma 21, every $[\![F]\!](\rho)$ is saturated.

Lemma 23 (Monotonicity of $[\![F]\!]$). *If*

$$\rho_1 \subseteq \rho_2 \qquad \textit{(pointwise)}$$

And F is a well-formed type whose free variables match the domains of ρ_1 and ρ_2, then

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

Proof. Observe that $\mathcal{A} \to (-)$ is a monotone operator.

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

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

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

If $\theta = N_1/y_1, ..., N_n/y_n$ and $\sigma = M_1/x_1, ..., M_m/x_m$ are simultaneous substitutions, we write $[\theta; \sigma]M$ to mean substituting the N_i with next substitution $[N_i/y_i]^\bullet(-)$ and the M_i with the current substitution $[M_i/x_i](-)$. We may write this explicitly as follows:

$$[N_1/y_1^\bullet, ..., N_n/y_n^\bullet; M_1/x_1, ..., M_m/x_m]M$$

If $\sigma = M_1/x_1, ..., M_m/x_m$ and $\mathbf{\Gamma} = x_1 : \mathcal{A}_1, ..., x_m : \mathcal{A}_m$, then we write $\sigma \in \mathbf{\Gamma}^\alpha$ to mean $M_i : \mathcal{A}_i^\alpha$ for all i, and similarly for $\theta \in \mathbf{\Theta}^\alpha$.

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

Lemma 26 (Semantic substitution lemma).
If $\mathbf{\Theta}; \mathbf{\Gamma}, x : \mathcal{A} \vDash M : \mathcal{B}$ and $\mathbf{\Theta}; \mathbf{\Gamma} \vDash N : \mathcal{A}$
then $\mathbf{\Theta}; \mathbf{\Gamma} \vDash [N/x]M : \mathcal{B}$

Proof.

1. For any α, substitutions $\theta \in (\blacktriangleright \mathbf{\Theta})^\alpha$ $\sigma \in \mathbf{\Gamma}^\alpha$ and $N' \in \mathcal{A}^\alpha$ we have $[\theta;\sigma,N'/x]M \in \mathcal{B}^\alpha$ (assumption)
2. For any α, and $\theta \in (\blacktriangleright \mathbf{\Theta})^\alpha$, $\sigma \in \mathbf{\Gamma}^\alpha$ we have $[\theta;\sigma]N \in \mathcal{A}^\alpha$ (assumption)
3. So taking $N' = [\theta;\sigma]N$ in line 1, we have $[\theta;\sigma,[\theta;\sigma]N/x]M \in \mathcal{B}^\alpha$
4. Hence $[\theta;\sigma]([N/x]M) \in \mathcal{B}^\alpha$ (since $[\theta;\sigma,[\theta;\sigma]N/x]M = [\theta;\sigma]([N/x]M)$)

□

Lemma 27. *If $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{A} \vDash \mathcal{H}[x] : \mathcal{C}$ where $\mathcal{C}$ is saturated and $\mathcal{H}$ is a head context not containing x, then $\mathbf{\Theta};\mathbf{\Gamma},x:\overline{\mathcal{A}} \vDash \mathcal{H}[x] : \mathcal{C}$*

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

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

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

Hence $\overline{\mathcal{A}}^\alpha \subseteq (\mathcal{C} \circ ([\theta;\sigma]\mathcal{H}))^\alpha$ (by Lemmas 19,11).

By definition, this is what we need to prove. □

Lemma 28. *If $\mathcal{A}$ is saturated and $\mathbf{\Theta};\mathbf{\Gamma} \vDash M' : \mathcal{A}$ and $\forall \alpha, \theta \in (\blacktriangleright \Theta)^\alpha, \sigma \in \mathbf{\Gamma}^\alpha$ we have $[\theta;\sigma]M \rightharpoonup^*_\alpha [\theta;\sigma]M'$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A}$*

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 used in the next lemma pertaining to map.

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

1. $\mathbf{\Theta};\mathbf{\Gamma} \vDash () : \mathbf{1}$
2. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A}$ and $\mathbf{\Theta};\mathbf{\Gamma} \vDash N : \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash (M,N) : \mathcal{A} \times \mathcal{B}$*
3. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A} \times \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{fst}\,M : \mathcal{A}$*
4. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A} \times \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{snd}\,M : \mathcal{B}$*
5. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inl}\,M : \mathcal{A} + \mathcal{B}$*
6. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inr}\,M : \mathcal{A} + \mathcal{B}$*
7. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A} + \mathcal{B}$ and $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{A} \vDash N_1 : \mathcal{C}$ and $\mathbf{\Theta};\mathbf{\Gamma},y:\mathcal{B} \vDash N_2 : \mathcal{C}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{case}\,M\,\mathsf{of}\,\mathsf{inl}\,x \mapsto N_1 \mid \mathsf{inr}\,y \mapsto N_2 : \mathcal{C}$*
8. *If $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{A} \vDash M : \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \lambda x.M : \mathcal{A} \to \mathcal{B}$*
9. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A} \to \mathcal{B}$ and $\mathbf{\Theta};\mathbf{\Gamma} \vDash N : \mathcal{A}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash MN : \mathcal{B}$*
10. *If $\cdot;\mathbf{\Theta} \vDash M : \mathcal{A}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \bullet M : \bigcirc\mathcal{A}$*
11. *If $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \bigcirc\mathcal{A}$ and $\mathbf{\Theta},x{:}\mathcal{A};\mathbf{\Gamma} \vDash N : \mathcal{B}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{let}\ \bullet x = M\ \mathsf{in}\,N : \mathcal{B}$*
12. *If $\mathcal{F}$ is a monotone function from saturated sets to saturated sets, and $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{F}(\boldsymbol{\mu}\mathcal{F})$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inj}\ M : \boldsymbol{\mu}\mathcal{F}$*
13. *If $\mathcal{F}$ is a monotone function from saturated sets to saturated sets, and $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \boldsymbol{\nu}\mathcal{F}$ then $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{out}\ M : \mathcal{F}(\boldsymbol{\nu}\mathcal{F})$*

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

2. (a) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A}$ (assumption)
   (b) $\mathbf{\Theta};\mathbf{\Gamma} \vDash N : \mathcal{B}$ (assumption)
   (c) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{fst}(M,N) : \mathcal{A}$ (by $\rightharpoonup$ closure)
   (d) $\mathbf{\Theta};\mathbf{\Gamma} \vDash (M,N) : \mathcal{A} \circ \mathsf{fst}$
   (e) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{snd}(M,N) : \mathcal{B}$ (by $\rightharpoonup$ closure, line (b))
   (f) $\mathbf{\Theta};\mathbf{\Gamma} \vDash (M,N) : \mathcal{B} \circ \mathsf{snd}$
   (g) $\mathbf{\Theta};\mathbf{\Gamma} \vDash (M,N) : \mathcal{A} \circ \mathsf{fst} \cap \mathcal{B} \circ \mathsf{snd}$ (lines (d),(f))
7. (a) $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{A} \vDash N_1 : \mathcal{C}$ (assumption)
   (b) $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{B} \vDash N_2 : \mathcal{C}$ (assumption)
   (c) $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{A} \vDash \mathsf{case}\ \mathsf{inl}\,x\,\mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$ (by $\rightharpoonup$ closure)
   (d) $\mathbf{\Theta};\mathbf{\Gamma},z:\mathcal{A} \star \mathsf{inl} \vDash \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$
   (e) $\mathbf{\Theta};\mathbf{\Gamma},x:\mathcal{B} \vDash \mathsf{case}\ \mathsf{inr}\,x\,\mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$ (by $\rightharpoonup$ closure)
   (f) $\mathbf{\Theta};\mathbf{\Gamma},z:\mathcal{B} \star \mathsf{inr} \vDash \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$
   (g) $\mathbf{\Theta};\mathbf{\Gamma},z:(\mathcal{A}\star\mathsf{inl} \cup \mathcal{B}\star\mathsf{inr}) \vDash \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$ (by lines (d),(f))
   (h) $\mathbf{\Theta};\mathbf{\Gamma},z : \overline{(\mathcal{A}\star\mathsf{inl} \cup \mathcal{B}\star\mathsf{inr})} \vDash \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$
   (i) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{A} + \mathcal{B}$ (assumption)
   (j) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x \mapsto N_1 \mid \mathsf{inr}\ y \mapsto N_2 : \mathcal{C}$ (subst. lemma, lines (h),(i))
10. We are given $\cdot;\mathbf{\Theta} \vDash M : \mathcal{A}$, i.e. for any α and $\theta \in \mathbf{\Theta}^\alpha$, that $[\cdot;\theta]M \in \mathcal{A}^\alpha$.
    Suppose we are given α and $\theta \in (\blacktriangleright \mathbf{\Theta})^\alpha$ and $\sigma \in \mathbf{\Gamma}^\alpha$.
    Case $\alpha = 0$: Then $(\blacktriangleright \mathcal{A})^0 = tm$, so $[\cdot;\theta]M \in (\blacktriangleright \mathcal{A})^0$
    Case $\alpha = m+1$: Then $(\blacktriangleright \mathcal{A})^{m+1} = \mathcal{A}^m$, and $\theta \in \mathbf{\Theta}^m$. By our assumption, taking $\alpha = m$, we have $[\cdot;\theta]M \in \mathcal{A}^m$
    Case $\alpha = \omega$: Then $(\blacktriangleright \mathcal{A})^\omega = \mathcal{A}^\omega$, and $\theta \in \mathbf{\Theta}^\omega$. By our assumption, $[\cdot;\theta]M \in \mathcal{A}^\omega$
    In any case, we have $[\cdot;\theta]M \in (\blacktriangleright \mathcal{A})^\alpha$.
    Then $[\theta;\sigma](\bullet M) = \bullet([\cdot;\theta]M) \in (\blacktriangleright \mathcal{A} \star \bullet)^\alpha$
    Hence $[\theta;\sigma](\bullet M) \in (\bigcirc\mathcal{A})^\alpha$ as required.
11. (a) $\mathbf{\Theta},x{:}\mathcal{A};\mathbf{\Gamma} \vDash N : \mathcal{C}$ (assumption)
    (b) $\mathbf{\Theta},x{:}\mathcal{A};\mathbf{\Gamma} \vDash \mathsf{let}\ \bullet x = \bullet x\ \mathsf{in}\ N : \mathcal{C}$ (by $\rightharpoonup$ closure)
    (c) $\mathbf{\Theta};\mathbf{\Gamma},y : \blacktriangleright \mathcal{A} \star \bullet \vDash \mathsf{let}\ \bullet x = y\ \mathsf{in}\ N : \mathcal{C}$
    (d) $\mathbf{\Theta};\mathbf{\Gamma},y : \overline{\blacktriangleright \mathcal{A} \star \bullet} \vDash \mathsf{let}\ \bullet x = y\ \mathsf{in}\ N : \mathcal{C}$ (lemma 27)
    (e) $\mathbf{\Theta};\mathbf{\Gamma},y : \bigcirc\mathcal{A} \vDash \mathsf{let}\ \bullet x = y\ \mathsf{in}\ N : \mathcal{C}$ (by def)
    (f) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \bigcirc\mathcal{A}$ (assumption)
    (g) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{let}\ \bullet x = M\ \mathsf{in}\ N : \mathcal{C}$ (subst lemma 26, lines (e),(f))
12. (a) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{F}(\boldsymbol{\mu}\mathcal{F})$ (assumption)
    (b) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inj}\ M : \mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \star \mathsf{inj}$
    (c) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inj}\ M : \overline{\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \star \mathsf{inj}}$ (monotonicity)
    (d) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inj}\ M : \mu(\mathcal{X} \mapsto \overline{\mathcal{F}(\mathcal{X}) \star \mathsf{inj}})$ (monotonicity, pre fixed point)
    (e) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{inj}\ M : \boldsymbol{\mu}\mathcal{F}$ (by def)
13. (a) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \boldsymbol{\nu}\mathcal{F}$ (assumption)
    (b) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \nu(\mathcal{X} \mapsto \mathcal{F}(\mathcal{X}) \circ \mathsf{out})$ (by def)
    (c) $\mathbf{\Theta};\mathbf{\Gamma} \vDash M : \mathcal{F}(\boldsymbol{\nu}\mathcal{F}) \circ \mathsf{out}$ (monotonicity, post fixed point)
    (d) $\mathbf{\Theta};\mathbf{\Gamma} \vDash \mathsf{out}\ M : \mathcal{F}(\boldsymbol{\nu}\mathcal{F})$ □

With this lemma, it remains only to handle map, iter, and coit. Below we show the semantic typing lemma for map. We write $\rho_1 \vDash \eta : \rho_2$ where $\eta = (x.M_1/X_1, ..., x.M_m/X_m)$ and $\rho_1 = \mathcal{A}_1/X_1, ..., \mathcal{A}_m/X_m$ and $\rho_2 = \mathcal{B}_1/X_1, ..., \mathcal{B}_m/X_m$ and the $\mathcal{A}_i$ and $\mathcal{B}_i$ are saturated to mean $x : \mathcal{A}_i \vDash M_i : \mathcal{B}_i$ for each i. Notably, because we define map independently of iter and coit, we can prove this directly before tackling iter and coit, offering a simplification of known proofs for interleaving $\mu\nu$. For readability, we write F instead of $(\Delta.F)$.

Lemma 30 (Semantic typing for map)**.** *If $\rho_1 \vDash \eta : \rho_2$ then $x : [\![F]\!](\rho_1) \vDash \mathsf{map}\ F\ \eta\ x : [\![F]\!](\rho_2)$*

Proof. Notice by unrolling the definitions, this is equivalent to showing $[\![F]\!](\rho_1) \subseteq [\![F]\!](\rho_2) \circ (\mathsf{map}\ \ F\ \eta)$. The proof proceeds by induction on F.

Case $F \times G$:

1. $x : [\![F]\!](\rho_1) \vDash \mathsf{map}\ F\ \eta\ x : [\![F]\!](\rho_2)$ (by I.H.)
2. $y : [\![F]\!](\rho_1) \times [\![G]\!](\rho_1) \vDash \mathsf{fst}\ y : [\![F]\!](\rho_1)$ (by lemma 29)
3. $y : [\![F]\!](\rho_1) \times [\![G]\!](\rho_1) \vDash \mathsf{map}\ F\ \eta\ (\mathsf{fst}\ y) : [\![F]\!](\rho_2)$ (by sem. subst 26)

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

Case $A \to F$:

1. $x : [\![F]\!](\rho_1) \vDash \mathsf{map}\ F\ \eta\ x : [\![F]\!](\rho_2)$ (by I.H.)
2. $y : [\![A]\!](\cdot) \to [\![F]\!](\rho_1), z : [\![A]\!] \vDash y\ z : [\![F]\!](\rho_1)$ (by lemma 29)
3. $y : [\![A]\!](\cdot) \to [\![F]\!](\rho_1), z : [\![A]\!] \vDash \mathsf{map}\ F\ \eta\ (y\ z) : [\![F]\!](\rho_2)$ (by sem. subst 26)
4. $y : [\![A]\!](\cdot) \to [\![F]\!](\rho_1) \vDash \lambda z.\ \mathsf{map}\ F\ \eta\ (y\ z) : [\![A]\!](\cdot) \to [\![F]\!](\rho_2)$ (by lemma 29)
5. $y : [\![A]\!](\cdot) \to [\![F]\!](\rho_1) \vDash \mathsf{map}\ (A \to F)\ \eta\ y : [\![A]\!](\cdot) \to [\![F]\!](\rho_2)$ (by $\twoheadrightarrow$ closure)
6. $y : [\![A \to F]\!](\rho_1) \vDash \mathsf{map}\ (A \to F)\ \eta\ y : [\![A \to F]\!](\rho_2)$ (by def)

Case $\bigcirc F$:

1. $x{:}[\![F]\!](\rho_1) \vDash \mathsf{map}\ F\ \eta\ x : [\![F]\!](\rho_2)$ (by I.H.)
2. $x{:}[\![F]\!](\rho_1); y{:}\ \bigcirc([\![F]\!](\rho_1)) \vDash \bullet(\mathsf{map}\ F\ \eta\ x) : \bigcirc([\![F]\!](\rho_2))$ (by lemma 29)
3. $y{:}\bigcirc([\![F]\!](\rho_1)) \vDash \mathsf{let}\ \bullet x = y\ \mathsf{in}\ \bullet(\mathsf{map}\ F\ \eta\ x) : \bigcirc([\![F]\!](\rho_2))$ (by lemma 29)
4. $y{:}[\![\bigcirc F]\!](\rho_1) \vDash \mathsf{let}\ \bullet x = y\ \mathsf{in}\ \bullet(\mathsf{map}\ F\ \eta\ x) : [\![\bigcirc F]\!](\rho_2)$ (by def)
5. $y{:}[\![\bigcirc F]\!](\rho_1) \vDash \mathsf{map}\ (\bigcirc F)\ \eta\ y : [\![\bigcirc F]\!](\rho_2)$ (by $\twoheadrightarrow$ closure)

Case $\mu X.F$: This proceeds primarily using the least fixed point property and $\twoheadrightarrow$ closure.
Let $\mathcal{C} = [\![\mu X.F]\!](\rho_2) = \mu(\mathcal{X} \mapsto \overline{[\![F]\!](\rho_2, \mathcal{X}) \star \mathsf{inj}})$
Let $\mathcal{D} = \mathcal{C} \circ (\mathsf{map}(\mu X.F)\eta)$
Let $N = (\mathsf{map}\ F\ (\eta, \mathsf{map}\ (\mu X.F)\ \eta))$

1. $\mathcal{D}$ is saturated (by Lemma 19)
2. $x : \mathcal{D} \vDash \mathsf{map}\ (\mu X.F)\ \eta\ x : \mathcal{C}$ (by definition of $\vDash$, $\mathcal{D}$)
3. $x : [\![F]\!](\rho_1, \mathcal{D}) \vDash \mathsf{map}\ F\ (\eta, \mathsf{map}\ (\mu X.F)\ \eta)\ x : [\![F]\!](\rho_2, \mathcal{C})$ (by I.H.)
4. $[\![F]\!](\rho_1, \mathcal{D}) \subseteq [\![F]\!](\rho_2, \mathcal{C}) \circ (\mathsf{map}\ F\ (\eta, \mathsf{map}\ (\mu X.F)\ \eta))$ (by definition of $\vDash$)
   $\subseteq [\![F]\!](\rho_2, \mathcal{C}) \star \mathsf{inj} \circ \mathsf{inj} \circ N$
   $\subseteq \overline{[\![F]\!](\rho_2, \mathcal{C}) \star \mathsf{inj}} \circ \mathsf{inj} \circ N$ (by mon. of $\circ$)
   $= \mathcal{C} \circ \mathsf{inj} \circ N$ (by rolling fixed point)
   $= \mathcal{C} \circ (\mathsf{inj}(\mathsf{map}\ F\ (\eta, \mathsf{map}\ (\mu X.F)\ \eta) -))$ (by def)
   $\subseteq \mathcal{C} \circ (\mathsf{map}\ (\mu X.F)\ \eta\ (\mathsf{inj}\ -))$ (by $\twoheadrightarrow$ closure)
   $= \mathcal{C} \circ (\mathsf{map}\ (\mu X.F)\ \eta) \circ \mathsf{inj}$
   $= \mathcal{D} \circ \mathsf{inj}$ (by def)
5. $[\![F]\!](\rho_1, \mathcal{D}) \star \mathsf{inj} \subseteq \mathcal{D}$ (by adjunction $\star$)
6. $\overline{[\![F]\!](\rho_1, \mathcal{D}) \star \mathsf{inj}} \subseteq \mathcal{D}$ (by adjunction $\overline{(-)}$)
7. $[\![\mu X.F]\!](\rho_1) \subseteq \mathcal{D}$ (by lfp property)
8. $y : [\![\mu X.F]\!](\rho_1) \vDash \mathsf{map}\ (\mu X.F)\ \eta\ y : [\![\mu X.F]\!](\rho_2)$ (by definitions of $\vDash$, $\mathcal{D}$, $\mathcal{C}$)

Case $\nu X.F$:
Let $\mathcal{C} = [\![\nu X.F]\!](\rho_1) = \nu(\mathcal{X} \mapsto [\![F]\!](\rho_1, \mathcal{X}) \circ \mathsf{out})$
Let $N = \mathsf{map}\ (\nu X.F)\ \eta$
Let $\mathcal{D} = \overline{\mathcal{C} \star N}$

1. $\mathcal{D}$ is saturated (by properties of $\overline{(-)}$, closure property of sn)
2. $x : \mathcal{C} \vDash Nx : \mathcal{D}$ (trivial, definition of $\mathcal{D}$)
3. $x : [\![F]\!](\rho_1, \mathcal{C}) \vDash \mathsf{map}\ F\ (\eta, N)\ x : [\![F]\!](\rho_2, \mathcal{D})$ (by I.H.)
4. $[\![F]\!](\rho_1, \mathcal{C}) \subseteq [\![F]\!](\rho_2, \mathcal{D}) \circ (\mathsf{map}\ F\ (\eta, N))$ (by definition of $\vDash$)
5. $\mathcal{C} \subseteq [\![F]\!](\rho_1, \mathcal{C}) \circ \mathsf{out}$ (unrolling fixed point)
   $\subseteq [\![F]\!](\rho_2, \mathcal{D}) \circ (\mathsf{map}\ F\ (\eta, N)) \circ \mathsf{out}$ (by mon. & line 5)
   $\subseteq [\![F]\!](\rho_2, \mathcal{D}) \circ \mathsf{out} \circ N$ (by $\twoheadrightarrow$ closure)
6. $\mathcal{C} \star N \subseteq [\![F]\!](\rho_2, \mathcal{D}) \circ \mathsf{out}$ (by $\star$-adjunction)
7. $[\![F]\!](\rho_2, \mathcal{D}) \circ \mathsf{out}$ is saturated (by Lemma 19)
8. $\mathcal{D} = \overline{\mathcal{C} \star N} \subseteq [\![F]\!](\rho_2, \mathcal{D}) \circ \mathsf{out}$ (by $\overline{(-)}$ adjunction, line 7)
9. $\mathcal{D} \subseteq [\![\nu X.F]\!](\rho_2)$ (by gfp property)
10. $\mathcal{C} \star N \subseteq [\![\nu X.F]\!](\rho_2)$ (monotonicity)
11. $\mathcal{C} \subseteq [\![\nu X.F]\!](\rho_2) \circ N$ (by $\star$-adjunction)
12. $[\![\nu X.F]\!](\rho_1) \subseteq [\![\nu X.F]\!](\rho_2) \circ (\mathsf{map}\ (\nu X.F)\ \eta)$ (by definitions of $\mathcal{C}$,N)
13. $y : [\![\nu X.F]\!](\rho_1) \vDash \mathsf{map}\ (\nu X.F)\ \eta\ y : [\![\nu X.F]\!](\rho_2)$
    (by definition of $\vDash$)

The other cases are similar. □

The only term constructors remaining to consider are iter and coit. These are the subject of the next two lemmas, which are proven similarly to the μ and ν cases of the map lemma. Namely, they proceed primarily by using the least (greatest) fixed point properties and backward closure under $\twoheadrightarrow$, appealing to the map lemma.

Lemma 31. *If $x : [\![F]\!](\mathcal{C}/X) \vDash M : \mathcal{C}$ where $\mathcal{C}$ is saturated, we have $y : [\![\mu X.F]\!] \vDash \mathsf{iter}_{X.F}\ (x.M)\ y{:}\ \mathcal{C}$*

Proof. Let $\mathcal{D} = \mathcal{C} \circ (\mathsf{iter}_{X.F}\ (x.M)\ )$

1. $\mathcal{D}$ is saturated (by Lemma 19)
2. $[\![F]\!](\mathcal{C}) \subseteq \mathcal{C} \circ ([-/x]M)$ (by assumption)
3. $\mathcal{D} \subseteq \mathcal{C} \circ (\mathsf{iter}_{X.F}(x.M))$ (by def)
4. $y : \mathcal{D} \vDash \mathsf{iter}_{X.F}\ (x.M)\ y : \mathcal{C}$ (by def of $\vDash$)
5. $y : [\![F]\!](\mathcal{D}) \vDash \mathsf{map}\ (X.F)\ (\mathsf{iter}_{X.F}\ (x.M))\ y : [\![F]\!](\mathcal{C})$ (by semantic typing of map, previous line)
6. $[\![F]\!](\mathcal{D}) \subseteq [\![F]\!](\mathcal{C}) \circ (\mathsf{map}(X.F)(\mathsf{iter}_{X.F}\ (x.M)))$ (by definition of $\vDash$)
   $\subseteq \mathcal{C} \circ ([-/x]M) \circ (\mathsf{map}(X.F)(\mathsf{iter}_{X.F}\ (x.M)))$ (by monotonicity of $\circ$, line 2)
   $= \mathcal{C} \circ ([(\mathsf{map}(X.F)(\mathsf{iter}_{X.F}\ (x.M)) -)/x]M)$
7. $[\![F]\!](\mathcal{D}) \subseteq sn \cap \mathcal{C} \circ ([(\mathsf{map}(X.F)(\mathsf{iter}_{X.F}\ (x.M)) -)/x]M)$ (previous line, saturated)
   $\subseteq \mathcal{C} \circ (\mathsf{iter}_{X.F}(x.M)(\mathsf{inj}\ -))$ (by $\twoheadrightarrow$ closure)
   $= \mathcal{C} \circ (\mathsf{iter}_{X.F}(x.M)) \circ \mathsf{inj}$
   $= \mathcal{D} \circ \mathsf{inj}$
8. $[\![F]\!](\mathcal{D}) \star \mathsf{inj} \subseteq \mathcal{D}$ (by adjunction $\star$)
9. $\overline{[\![F]\!](\mathcal{D}) \star \mathsf{inj}} \subseteq \mathcal{D}$ (by adjunction $\overline{(-)}$)
10. $\mu(\mathcal{X} \mapsto \overline{[\![F]\!](\mathcal{X}) \star \mathsf{inj}}\ ) \subseteq \mathcal{D}$ (lfp property)
11. $y : [\![\mu X.F]\!] \vDash \mathsf{iter}_{X.F}\ (x.M)\ y{:}\ \mathcal{C}$ (by definitions)

□

Corollary 32. *If $x : [\![F]\!](\mathcal{C}) \vDash M : \mathcal{C}$ and $\mathbf{\Theta}; \mathbf{\Gamma} \vDash N : [\![\mu X.F]\!]$ where $\mathcal{C}$ is saturated, then $\mathbf{\Theta}; \mathbf{\Gamma} \vDash \mathsf{iter}_{X.F}\ (x.M)\ N\ : \mathcal{C}$*

Lemma 33. *If $x : \mathcal{C} \vDash M : [\![F]\!](\mathcal{C}/X)$ where $\mathcal{C}$ is saturated, we have $y : \mathcal{C} \vDash \mathsf{coit}_{X.F}\ (x.M)\ y{:}\ [\![\nu X.F]\!]$*

Proof. Let $\mathcal{D} = \overline{\mathcal{C} \star (\mathsf{coit}_{X.F}\ (x.M)\ )}$

1. $\mathcal{D}$ is saturated (property of $\overline{(-)}$, closure property of sn)

2. $\mathcal{C} \subseteq \mathcal{C} \star (\mathsf{coit}_{X.F}\ (x.M)\ ) \circ (\mathsf{coit}_{X.F}\ (x.M)\ )$
$\subseteq \mathcal{D} \circ (\mathsf{coit}_{X.F}\ (x.M)\ )$ (monotonicity of $\circ$)
3. $y : \mathcal{C} \vDash \mathsf{coit}_{X.F}\ (x.M)\ y\ : \mathcal{D}$ (by def of $\vDash$)
4. $y : [\![F]\!](\mathcal{C}) \vDash \mathsf{map}(X.F)(\mathsf{coit}_{X.F}\ (x.M)\ )y : [\![F]\!](\mathcal{D})$ (semantic typing of map)
5. $[\![F]\!](\mathcal{C}) \subseteq [\![F]\!](\mathcal{D}) \circ (\mathsf{map}(X.F)(y.\ \mathsf{coit}_{X.F}\ (x.M)\ y))$ (definition of $\vDash$)
6. $\mathcal{C} \subseteq [\![F]\!](\mathcal{C}) \circ ([-/x]M)$ (by assumption)
$\subseteq [\![F]\!](\mathcal{D}) \circ (\mathsf{map}(X.F)(y.\ \mathsf{coit}_{X.F}\ (x.M)\ y)) \circ ([-/x]M)$ (monotonicity of $\circ$, previous line)
$= [\![F]\!](\mathcal{D}) \circ (\mathsf{map}(X.F)(y.\ \mathsf{coit}_{X.F}\ (x.M)\ y)([-/x]M))$
7. $\mathcal{C} \subseteq sn \cap [\![F]\!](\mathcal{D}) \circ (\mathsf{map}(X.F)(y.\ \mathsf{coit}_{X.F}\ (x.M)\ y)([-/x]M))$ (previous line, $\mathcal{C}$ saturated)
$\subseteq [\![F]\!](\mathcal{D}) \circ (\mathsf{out}(\mathsf{coit}_{X.F}\ (x.M)\ -))$ (by $\twoheadrightarrow$ closure)
$= [\![F]\!](\mathcal{D}) \circ \mathsf{out} \circ (\mathsf{coit}_{X.F}\ (x.M)\ )$
8. $\mathcal{C} \star (\mathsf{coit}_{X.F}\ (x.M)\ ) \subseteq [\![F]\!](\mathcal{D}) \circ \mathsf{out}$ (by adjunction $\star$)
9. $\mathcal{D} \subseteq [\![F]\!](\mathcal{D}) \circ \mathsf{out}$ (by adjunction $\overline{(-)}$)
10. $\mathcal{D} \subseteq \nu(\mathcal{X} \mapsto [\![F]\!](\mathcal{X}) \circ \mathsf{out})$ (by gfp property)
$= [\![\nu X.F]\!]$
11. $\mathcal{C} \star (\mathsf{coit}_{X.F}\ (x.M)\ ) \subseteq [\![\nu X.F]\!]$
12. $\mathcal{C} \subseteq [\![\nu X.F]\!] \circ (\mathsf{coit}_{X.F}\ (x.M)\ )$ (adjunction $\star$)
13. $y : \mathcal{C} \vDash \mathsf{coit}_{X.F}\ (x.M)\ y$: $[\![\nu X.F]\!]$ (by definitions)

$\square$

Corollary 34. *If $x : \mathcal{C} \vDash M : [\![F]\!](\mathcal{C})$ and $\mathbf{\Theta}; \mathbf{\Gamma} \vDash N : \mathcal{C}$ where $\mathcal{C}$ is saturated, then $\mathbf{\Theta}; \mathbf{\Gamma} \vDash \mathit{coit}_{X.F}\ (x.M)\ N\ : [\![\nu X.F]\!]$*

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 35 (Fundamental theorem)**.** *If $\Theta; \Gamma \vdash M : A$, then we have $[\![\Theta]\!]; [\![\Gamma]\!] \vDash M : [\![A]\!]$*

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

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

Similarly, we can demonstrate an interesting property of the type $B \equiv \mu X.\bigcirc X$. First, there is no closed term of type $B \to \bot$. For if there was some $f : B \to \bot$, we could normalize $x : B; \cdot \vdash f(\mathsf{inj}\ \bullet x) : B$ at time 0 to obtain a 0-value $x : B; \cdot \vdash v : \bot$, which cannot exist. Moreover, there is no closed term of type B, since there is no closed ω-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 $x_0 : \Diamond P \vdash M_0 : \Diamond Q$. 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 $Bool$ or $\mathbb{N}$. We consider sequences of interaction which begin as follows, where we write $\mathsf{now}\ p$ for $\mathsf{inj}\ (\mathsf{inl}\ p)$ and $\mathsf{later} \bullet t$ for $\mathsf{inj}\ (\mathsf{inr}(\bullet t))$.

$$\begin{array}{lcl} [\mathsf{later} \bullet x_1/x_0]M_0 & \rightsquigarrow_0^* & \mathsf{later} \bullet M_1 \\ [\mathsf{later} \bullet x_2/x_1]M_1 & \rightsquigarrow_0^* & \mathsf{later} \bullet M_2 \\ & \vdots & \end{array}$$

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 refine 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 $x_i{:}\Diamond P \vdash M_i : \Diamond Q$ 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:

$$[\mathsf{now}\ p/x_{i+1}]M_{i+1} \quad \rightsquigarrow_\omega^* \quad v \not\rightsquigarrow_\omega$$

In this case, $\vdash [\mathsf{now}\ p/x_{i+1}]M_{i+1} : \Diamond Q$. By type preservation and strong normalization, we can evaluate this completely to $\vdash v : \Diamond Q$. By an inspection of the closed ω-values, v must be of the form $\mathsf{later}(\bullet\mathsf{later}(\bullet \cdots (\mathsf{now}\ q)))$. 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:

$$\begin{array}{lcl} & \vdots & \\ [\mathsf{later} \bullet x_{i+1}/x_i]M_i & \rightsquigarrow_0^* & \mathsf{now}\ q \end{array}$$

We remark that type preservation of $\rightsquigarrow_0$ provides an explanation of *causality*: since $x_{i+1} : \Diamond P; \cdot \vdash [\mathsf{later} \bullet x_{i+1}/x_i]M_i : \Diamond Q$, if evaluating the term $[\mathsf{later} \bullet x_{i+1}/x_i]M_i$ with $\rightsquigarrow_0$ produces a 0-value v, then $x_{i+1} : \Diamond P; \cdot \vdash v : \Diamond Q$ and by an inspection of the 0-values of this type, we see that v must be of the form $\mathsf{later} \bullet M_{i+1}$ or $\mathsf{now}\ p$ – 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 $\rightsquigarrow_0$ guarantees *reactive productivity*. That is, the evaluation of $[\mathsf{later} \bullet x_{i+1}/x_i]M_i$ 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 [25], one could generalize this kind of argument to reduction at n, and normalize terms using $\rightsquigarrow_n$ 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 $\rightsquigarrow_\omega$ provides an explanation of *liveness*. When evaluating a closed term $\vdash M : \Diamond Q$ at ω, we arrive at a closed ω-value $v : \Diamond Q$. 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 prequisite 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 [24–26]. 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 [16] *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, 26], 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. $\Box A$), 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) μ and ν, 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 Sec. 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 or metatheory as we do here. His CPCs may provide a promising denotational semantics for our calculus.

The classical linear μ-calculus (often called νTL) forms the inspiration for our calculus. The study of (classical) νTL proof systems goes back at least to Lichtenstein [27]. Our work offers a constructive, type-theoretic view on νTL.

Synchronous dataflow languages, such as Esterel [6], Lustre [9], and Lucid Synchrone [35] 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 μ-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 `inj` 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 μ and ν 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 $\to$ and ν. 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 μ and ν is challenging – as Altenkirch and Danielsson [5] 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 [25]) – roughly, types built with only ν, not μ. 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. [3] in this setting. This needs a careful treatment, because matching under a $\bullet$ 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 ω, 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

[1] A. Abel. Mixed inductive/coinductive types and strong normalization. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS'07, pages 286–301. Springer-Verlag, 2007.

[2] A. Abel and T. Altenkirch. A predicative strong normalisation proof for a lambda-calculus with interleaving inductive types. In *TYPES*, pages 21–40, 1999.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38. ACM, 2013.

[4] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[5] T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, 2010.

[6] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, UK, 1985. Springer-Verlag. ISBN 3-540-15670-4.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[8] L. Birkedal, R. E. Mogelberg, J. Schwinghammer, and K. Stovring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55 –64. IEEE Computer Society, 2011.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188. ACM, 1987.

[10] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308. Springer-Verlag, 2006.

[11] A. Courtney. Frappé functional reactive programming in java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44. Springer-Verlag, 2001.

[12] J. Donham. Functional reactive programming in OCaml. URL `https://github.com/jaked/froc`.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

[14] J. Y. Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmtique d'ordre supérieur*. These d'état, Université de Paris 7, 1972.

[15] J.-Y. Girard, Y. Lafont, and P. Tayor. *Proofs and types*. Cambridge University Press, 1990.

[16] H. Hiroshi Nakano. A modality for recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 255 –266. IEEE Computer Society, 2000.

[17] A. Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth Workshop on Programming Languages meets Program Verification*, pages 49–60. ACM, 2012.

[18] A. Jeffrey. Causality for free! parametricity implies causality for functional reactive programs. In *Proceedings of the seventh Workshop on Programming Languages meets Program Verification*, pages 57–68. ACM, 2013.

[19] A. Jeffrey. Functional reactive programming with liveness guarantees. In *18th International Confernece on Functional Programming*. ACM, 2013.

[20] W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286(0):229–242, 2012.

[21] W. Jeltsch. Temporal logic with until: Functional reactive programming with processes and concrete process categories. In *Proceedings of the seventh Workshop on Programming Languages meets Program Verification*, pages 69–78. ACM, 2013.

[22] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349 – 391, 1997.

[23] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333 – 354, 1983.

[24] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *18th International Confernece on Functional Programming*. ACM, 2013.

[25] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266. IEEE Computer Society, 2011.

[26] N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 45–58. ACM, 2012.

[27] O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, The Weizmann Institute of Science, 1991.

[28] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[29] R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, University of Munich, 1998.

[30] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1988.

[31] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. ACM, 2002.

[32] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. Technical Report 33D.

[33] D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science, Proceedings of the 5th GI-Conference*, Lecture Notes in Computer Science (LNCS 104, pages 167–183. Springer, 1981.

[34] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[35] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[36] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967.