Productive Coprogramming with Guarded Recursion

Robert Atkey
bob.atkey@gmail.com

Conor McBride
University of Strathclyde
Conor.McBride@strath.ac.uk

Abstract

Total functional programming offers the beguiling vision that, just by virtue of the compiler accepting a program, we are guaranteed that it will always terminate. In the case of programs that are not intended to terminate, e.g., servers, we are guaranteed that programs will always be *productive*. Productivity means that, even if a program generates an infinite amount of data, each piece will be generated in finite time. The theoretical underpinning for productive programming with infinite output is provided by the category theoretic notion of final coalgebras. Hence, we speak of *co*programming with non-well-founded *co*data, as a dual to programming with well-founded data like finite lists and trees.

Systems that offer facilities for productive coprogramming, such as the proof assistants Coq and Agda, currently do so through syntactic guardedness checkers, which ensure that all self-recursive calls are guarded by a use of a constructor. Such a check ensures productivity. Unfortunately, these syntactic checks are not compositional, and severely complicate coprogramming.

Guarded recursion, originally due to Nakano, is tantalising as a basis for a flexible and compositional type-based approach to coprogramming. However, as we show, guarded recursion by itself is not suitable for coprogramming due to the fact that there is no way to make finite observations on pieces of infinite data. In this paper, we introduce the concept of *clock variables* that index Nakano's guarded recursion. Clock variables allow us to "close over" the generation of infinite codata, and to make finite observations, something that is not possible with guarded recursion alone.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (functional) programming; D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Languages, Theory, Types, Recursion

Keywords coalgebras, corecursion, guarded recursion, total functional programming

1. Introduction

Coprogramming refers to the practice of explicitly manipulating codata, the non-well-founded dual of well-founded data like finite lists and trees. Codata are useful for representing the output of an infinite process as a never ending stream, or for representing potentially infinite search trees of possibilities. The natural way to express codata is as a recursively defined coprogram. However, for recursively defined coprograms to be safely used in a total setting, the system must ensure that all recursive definitions are productive. The state of the art for productivity checking is currently either awkward syntactic guardedness checkers, or more flexible sized type systems that explicitly label everything with size information.

In this paper, we investigate the use of *guarded recursion* (due to Hiroshi Nakano [20]) as a lightweight typing discipline for enabling productive coprogramming. As we show in this introduction, guarded recursion by itself is not suitable for coprogramming, due to the strict segmentation of time inherent in putting guardedness information in types. We therefore introduce the concept of *clock variables* to take us from the finite-time-sliced world of guarded recursion to the infinite world of codata.

The contributions we make in this paper are the following:

1. We define a core type system for comfortable productive coprogramming, combining three key features: initial algebras, Nakano's guarded recursion and our main conceptual contribution: quantification over clock variables. We show that by combining the three key features of our system, we obtain a system for programming with *final coalgebras*, the category theoretic description of codata. The combination of guarded recursion and clock quantification allows us to dispense with the clunky syntactic guardedness checks, and move to a flexible, compositional and local type-based system for ensuring guardedness.
2. We define a domain-theoretic denotational model that effectively interprets our system in the untyped lazy λ-calculus. We use a multiply-step-indexed model of types to prove that all well-typed programs are either terminating or productive and to show the correctness of our reconstruction of final coalgebras.
3. A side benefit of the combination of guarded recursion and clock quantification is that, due to the careful tracking of what data is available when through guarded types, we are able to accurately type, and show safe, strange circular functional programs like Bird's replaceMin example.

Despite the large amount of recent work on guarded recursion (we provide references throughout this introduction), this is the first paper that formally links guarded recursion to productive coprogramming. As recently noted by Birkedal and Møgelberg [4], "guarded recursive types can be used to [sic] as a crutch for higher-order programming with coinductive types". In this paper, we show that this is indeed possible.

1.1 Guardedness checkers and guarded recursion

Several systems with support for corecursion, such as Coq (as described by Giménez [12]) and Agda (as described by Danielsson and Altenkirch [9]), make use of syntactic guardedness checks to

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500597

ensure productivity of programs defined using corecursion. Without these checks, the soundness of these systems cannot be guaranteed. An unfortunate aspect of these guardedness checks is their non-compositionality. We demonstrate the problem with an example, and see how Nakano's guarded recursion can be used to provide a compositional type-based guardedness check.

We will use Haskell notation for each of our informal examples, since it serves to illustrate the issues concisely. Consider the following Haskell declaration of a type of infinite streams of integers:

$$\textbf{data}\ \text{Stream} = \text{StreamCons}\ \text{Integer}\ \text{Stream}$$

An example of a Stream is the infinite stream of 1s:

$$\begin{aligned} &\mathit{ones} :: \text{Stream} \\ &\mathit{ones} = \text{StreamCons}\ 1\ \mathit{ones} \end{aligned}$$

It is easy to see that this recursive definition is *guarded*; *ones* is only invoked recursively within an application of the stream constructor StreamCons. This definition of *ones* defines an infinite data structure, but each piece of the data structure is delivered to us in finite time. Hence, *ones* is productive.

An example of a non-guarded definition is the filter function, extended from finite lists to infinite streams:

$$\begin{aligned} &\mathit{filter} :: (\text{Integer} \to \text{Bool}) \to \text{Stream} \to \text{Stream} \\ &\mathit{filter}\ f\ (\text{StreamCons}\ z\ s) = \\ &\quad \textbf{if}\ f\ z\ \textbf{then}\ \text{StreamCons}\ z\ (\mathit{filter}\ f\ s)\ \textbf{else}\ \mathit{filter}\ f\ s \end{aligned}$$

This definition is not guarded: in the **then**-case of the conditional, the recursive call to *filter* is not within an application of the stream constructor StreamCons. A syntactic guardedness checker is right to reject this definition: consider the case when all of the elements of the stream are filtered out; *filter* will never return anything, and so will be non-productive.

Syntactic guardedness checkers are not always so helpful. The following higher-order function defines a general merge function on pairs of streams:

$$\begin{aligned} &\mathit{mergef} :: (\text{Integer} \to \text{Integer} \to \text{Stream} \to \text{Stream}) \to \\ &\qquad \text{Stream} \to \text{Stream} \to \text{Stream} \\ &\mathit{mergef}\ f\ (\text{StreamCons}\ x\ xs)\ (\text{StreamCons}\ y\ ys) = \\ &\quad f\ x\ y\ (\mathit{mergef}\ f\ xs\ ys) \end{aligned}$$

Any syntactic checker looking for constructors to guard recursive calls will reject this function: there are no constructors anywhere in the definition! This rejection is with good reason: there are functions that we could pass to *mergef* that would render it unproductive. For example, this function will cause *mergef* to hang on all pairs of streams:

$$\begin{aligned} &\mathit{badf} :: \text{Integer} \to \text{Integer} \to \text{Stream} \to \text{Stream} \\ &\mathit{badf}\ x\ y\ s = s \end{aligned}$$

On the other hand, there are plenty of good functions *f* that we could use with *mergef* to obtain productive functions, but a syntactic guardedness checker does not allow us to express this fact.

A possible way out of this problem is to retain the syntactic guardedness check, and work around it by changing the type of *f*. For instance, we could change the required type of *f* to:

$$f :: \text{Integer} \to \text{Integer} \to \text{Integer}$$

to allow for merging functions that operate element-wise. Or we could change the type to

$$f :: \text{Integer} \to \text{Integer} \to (\text{Integer}, [\text{Integer}])$$

which would allow for the functional argument to replace each pair of elements from the input streams with a non-empty list of values. The general trick is to make *f* return "instructions" on how to transform the stream back to *mergef*, which then executes them in a way that the syntactic guardedness checker can see is guarded. This technique has been elaborated in detail by Danielsson [8]. However, it is difficult to see whether we could capture all the possibilities for good *f*s in a single type. We could give yet another type which would accommodate the behaviour of the following possible value of *f*:

$$f\ x\ y\ s = \text{StreamCons}\ x\ (\mathit{map}\ (+y)\ s)$$

Incorporating the forms of all the possible productive definitions into a single type seems impractical. Moreover, we complicate the definition of *mergef*, which is now forced to become essentially an implementation of a virtual machine for running little stream transformer programs returned by its functional argument, rather than the simple one line definition we gave above.

A more promising type-based solution is provided by Nakano [20]. Nakano introduces a guardedness type constructor that represents values that may only be used in a guarded way. Thus Nakano transports the guardedness checks from the syntactic level into the type system. We write this constructor, applied to a type A, as $\triangleright A$ (Nakano uses the notation $\bullet A$, but the $\triangleright A$ notation has become more common, and conveys an intuitive idea of displacement in time). A useful way to think of $\triangleright A$ is as a value of A that is only available "tomorrow", and the temporal gap between today and tomorrow can only be bridged by the application of a constructor such as StreamCons. The reading of $\triangleright A$ as a value of A tomorrow is explicitly supported in the step-indexed semantics we present in Section 3 by means of a clock counting down to zero.

We now alter the type of *f* to be:

$$f :: \text{Integer} \to \text{Integer} \to \triangleright\text{Stream} \to \text{Stream}$$

This type captures the intuition that the stream argument to *f* must only be used in a guarded way. To introduce guarded types into the system, we must make some changes to the types of our primitives. We alter the type of the constructor StreamCons as follows, and also introduce a stream deconstructor that we previously implicitly used via pattern matching:

$$\begin{aligned} &\text{StreamCons} :: \text{Integer} \to \triangleright\text{Stream} \to \text{Stream} \\ &\text{deStreamCons} :: \text{Stream} \to (\text{Integer}, \triangleright\text{Stream}) \end{aligned}$$

The type of StreamCons shows that it takes a guarded stream "tomorrow", and produces a non-guarded stream "today". This is in line with the temporal intuition we have of the type $\triangleright$Stream. The inverse operation deStreamCons takes a full stream and returns the integer and the *guarded* remainder of the stream.

To give the guardedness type constructor force, Nakano proposes the following alternative type of the fixpoint operator for defining recursive values:

$$\textbf{fix} :: (\triangleright A \to A) \to A$$

(The standard typing of the **fix** operator is $(A \to A) \to A$.) Nakano's alternative typing ensures that recursive definitions must be guarded by only allowing access to recursively generated values "tomorrow".

To actually program with the guardedness constructor, we equip it with the structure of an applicative functor [18]:

$$\begin{aligned} &\textbf{pure} :: A \to \triangleright A \\ &(\circledast) :: \triangleright(A \to B) \to \triangleright A \to \triangleright B \end{aligned}$$

Note that $\triangleright$ is not a monad: a join operation of type $\triangleright(\triangleright A) \to \triangleright A$ would allow us to collapse multiple guardedness obligations.

This method for integrating Nakano's guardedness type constructor into a typed λ-calculus is not the only one. Nakano uses a system of subtyping with respect to the guardedness type constructor that has a similar effect to assuming that $\triangleright$ is an applicative functor. We propose to treat $\triangleright$ as an applicative functor here, due to the greater ease with which it can be incorporated into a standard functional programming language like Haskell. Krishnaswami and

Benton [14, 15] have presented an alternative method that annotates members of the typing context with the level of guardedness that applies to them. Severi and de Vries [22] have generalised Krishnaswami and Benton's approach for any typed λ-calculus derived from a Pure Type System (PTS), including dependently typed systems. These calculi have nice proof theoretic properties, such as being able to state productivity in terms of infinitary strong normalisation.

With the guardedness type constructor, and its applicative functor structure, we can rewrite the *mergef* function to allow the alternative typing which only allows functional arguments that will lead to productive definitions.

$$\begin{aligned}&\mathit{mergef} :: (\text{Integer} \to \text{Integer} \to \triangleright\text{Stream} \to \text{Stream}) \to \\ &\qquad \text{Stream} \to \text{Stream} \to \text{Stream}\\ &\mathit{mergef}\, f = \mathbf{fix}\ (\lambda g\ xs\ ys.\\ &\qquad\qquad \mathbf{let}\ (x, xs') = \mathsf{deStreamCons}\ xs\\ &\qquad\qquad\quad\ (y, ys') = \mathsf{deStreamCons}\ ys\\ &\qquad\qquad \mathbf{in}\ f\, x\, y\ (g \circledast xs' \circledast ys'))\end{aligned}$$

Aside from the explicit uses of **fix** and deStreamCons, which were hidden by syntactic sugar in our previous definition, the only substantial changes to the definition are the uses of applicative functor apply ($\circledast$). These uses indicate operations that are occurring under a guardedness type constructor.

1.2 From the infinite to the finite

The type system for guarded recursion that we described above allowed us to remove the syntactic guardedness check and replace it with a compositional type-based one. However, aside from proposing applicative functor structure as a convenient way to incorporate the guardedness type constructor into a functional language, we have not gone much beyond Nakano's original system. We now describe a problem with guarded recursion when attempting to combine infinite and finite data. We propose a solution to this problem in the next subsection.

Consider the *take* function that reads a finite prefix of an infinite stream into a list. This function will have the following type:

$$\mathit{take} :: \text{Natural} \to \text{Stream} \to [\text{Integer}]$$

where we wish to regard the type [Integer] as the type of *finite* lists of integers. The explicit segregation of well-founded and non-well-founded types is important for our intended applications of total functional programming and theorem proving.

We also assume that the type Natural of natural numbers is well-founded, so we may attempt to write *take* by structural recursion on its first argument. However, we run into a difficulty, which we have highlighted.

$$\begin{aligned}&\mathit{take} :: \text{Natural} \to \text{Stream} \to [\text{Integer}]\\ &\mathit{take}\ 0\ s = []\\ &\mathit{take}\ (n+1)\ s = x : \mathit{take}\ n\ s'\\ &\quad \mathbf{where}\ (x, s') = \mathsf{deStreamCons}\ s\end{aligned}$$

The problem is that the variable s', which we have obtained from deStreamCons, has type $\triangleright$Stream. However, to invoke the *take* function structurally recursively, we need something of type Stream, without the guardedness restriction.

We analyse this problem in terms of our intuitive reading of $\triangleright$Stream as a stream that is available "tomorrow". Nakano's typing discipline slices the construction of infinite data like Stream into discrete steps. In contrast, a well-founded data type like [Integer] lives entirely in the moment. While a stream is being constructed, this slicing is required so that we do not get ahead of ourselves and attempt to build things today from things that will only be available tomorrow. But once a stream has been fully constructed, we ought to be able to blur the distinctions between the days of its construction. We accomplish this in our type system by means of *clock variables*, and quantification over them.

1.3 Clock variables

We extend the type system with *clock variables* κ. A clock variable κ represents an individual time sequence that can be used for safe construction of infinite data like streams. In our model, clock variables are interpreted as counters running down to zero in the style of step-indexed models. By quantifying over all counters we are able to accommodate all possible finite observations on an infinite data structure.

We annotate the guardedness type constructor with a clock variable to indicate which time stream we are considering: $\triangleright^\kappa A$. Infinite data types such as streams are now annotated by their clock variable, indicating the time stream that they are being constructed on. We have the following new types for the constructor and deconstructor:

$$\begin{aligned}&\mathsf{StreamCons}^\kappa :: \text{Integer} \to \triangleright^\kappa\text{Stream}^\kappa \to \text{Stream}^\kappa\\ &\mathsf{deStreamCons}^\kappa :: \text{Stream}^\kappa \to (\text{Integer}, \triangleright^\kappa\text{Stream}^\kappa)\end{aligned}$$

We now regard the type Stream^κ as the type of infinite streams in the process of construction. A finished infinite stream is represented by *quantifying* over the relevant clock variable: $\forall\kappa.\text{Stream}^\kappa$. If we think of each possible downward counting counter that κ could represent, then this universally quantified type allows for any counter large enough to allow us any finite extraction of data from the stream. We use the notation $\Lambda\kappa.e$ and $e[\kappa]$ to indicate clock abstraction and application respectively, in the style of explicitly typed System F type abstraction and application.

The fixpoint operator is parameterised by the clock variable:

$$\mathbf{fix} :: \forall\kappa.(\triangleright^\kappa A \to A) \to A$$

as is the applicative functor structure carried by $\triangleright^\kappa$:

$$\begin{aligned}&\mathbf{pure} :: \forall\kappa.A \to \triangleright^\kappa A\\ &(\circledast) :: \forall\kappa.\triangleright^\kappa(A \to B) \to \triangleright^\kappa A \to \triangleright^\kappa B\end{aligned}$$

We also add an additional piece of structure to eliminate guarded types that have been universally quantified:

$$\mathbf{force} :: (\forall\kappa.\ \triangleright^\kappa A) \to (\forall\kappa.\ A)$$

Intuitively, if we have a value that, for any time stream, is one time step away, we simply instantiate it with a time stream that has at least one step left to run and extract the value. Note that the universal quantification is required on the right hand side since κ may appear free in the type A. The **force** operator has a similar flavour to the **runST** $:: (\forall s.\text{ST}\ s\ a) \to a$ for the ST monad [16, 19]. In both cases, rank-2 quantification is used to prevent values from "leaking" out from the context in which it is safe to use them.

Using **force** we may write a deconstructor for completed streams of type $\forall\kappa.\text{Stream}^\kappa$.

$$\begin{aligned}&\mathsf{deStreamCons} :: (\forall\kappa.\text{Stream}^\kappa) \to (\text{Integer}, \forall\kappa.\text{Stream}^\kappa)\\ &\mathsf{deStreamCons}\ x =\\ &\qquad (\Lambda\kappa.\mathbf{fst}\ (\mathsf{deStreamCons}^\kappa\ (x[\kappa])),\\ &\qquad\ \mathbf{force}\ (\Lambda\kappa.\mathbf{snd}\ (\mathsf{deStreamCons}^\kappa\ (x[\kappa]))))\end{aligned}$$

In this definition we have made use of a feature of our type system that states that the type equality $\forall\kappa.A \equiv A$ holds whenever κ is not free in A. Our system also includes other type equalities that demonstrate how clock quantification interacts with other type formers. These are presented in Section 2.2.

In the presence of clock variables, our definition of *take* is well-typed and we get the function we desire: taking a finite observation

of a piece of infinite data:

$$\begin{aligned}&\mathit{take} :: \text{Natural} \to (\forall\kappa.\text{Stream}^{\kappa}) \to [\text{Integer}]\\&\mathit{take}\ 0\ s = []\\&\mathit{take}\ (n+1)\ s = x : \mathit{take}\ n\ s'\\&\quad \textbf{where}\ (x, s') = \mathsf{deStreamCons}\ s\end{aligned}$$

Clock variables also allow us to make clear in the types fine distinctions between functions that are extensionally equal, but differ in their productivity. In plain Haskell, the following two stream mapping functions have the same behaviour on any *completely constructed* stream. The first *map* processes each element one at a time

$$\begin{aligned}&\mathit{map}\ f\ s = \mathsf{StreamCons}\ (f\ x)\ (\mathit{map}\ f\ s')\\&\quad \textbf{where}\ (x, s') = \mathsf{deStreamCons}\ s\end{aligned}$$

while *maap* processes elements two at a time by pattern matching:

$$\begin{aligned}&\mathit{maap}\ f\ (\mathsf{StreamCons}\ x\ (\mathsf{StreamCons}\ y\ s''))\\&\quad = \mathsf{StreamCons}\ (f\ x)\ (\mathsf{StreamCons}\ (f\ y)\ (\mathit{maap}\ f\ s''))\end{aligned}$$

If all the elements of the input are available at once, then *map* and *maap* have the same behaviour. However, if these functions are passed streams that are still being constructed, then their behaviour can differ significantly. Consider these two definitions of a stream of natural numbers, using *map* and *maap*:

$$\begin{aligned}&\mathit{nats} = \mathsf{StreamCons}\ 0\ (\mathit{map}\ (\lambda x.\ x+1)\ \mathit{nats})\\&\mathit{badnats} = \mathsf{StreamCons}\ 0\ (\mathit{maap}\ (\lambda x.\ x+1)\ \mathit{badnats})\end{aligned}$$

Running *nats* produces the infinite stream of natural numbers, as expected, but running *badnats* produces nothing. The function *maap* expects two elements to be available on its input stream, while *badnats* has only provided one.

Even though *map* and *maap* are extensionally equal, they have different behaviour on partially constructed streams. We can state this different behaviour using the following types in our system:

$$\begin{aligned}&\mathit{map} :: \forall\kappa.(\text{Integer} \to \text{Integer}) \to \text{Stream}^{\kappa} \to \text{Stream}^{\kappa}\\&\mathit{maap} :: (\text{Integer} \to \text{Integer}) \to (\forall\kappa.\text{Stream}^{\kappa}) \to (\forall\kappa.\text{Stream}^{\kappa})\end{aligned}$$

Thus, *map*'s type states that the input stream and output stream run on the same clock κ, so each element of the output will only require one element of the input. This is exactly what *nats* requires, and our type system will accept the definition of *nats*. In contrast, *maap*'s type states that it requires a fully constructed stream as input. The definition of *badnats* does not provide this, and so our type system will reject it.

By using clock quantification to "close over" the guardedness, our approach thus *localises* the discipline which makes the productivity of definitions plain, without affecting the types at which the defined objects are used. Abel's *sized types* impose a similar discipline, but globally [1].

1.4 Final coalgebras from initial algebras, guards and clocks

In the previous subsection, we informally stated that the type $\text{Stream} = \forall\kappa.\text{Stream}^{\kappa}$ represents exactly the type of infinite streams of integers. We make this informal claim precise by using the category theoretic notion of final coalgebra. A key feature of our approach is that we decompose the notion of final coalgebra into three parts: initial algebras, Nakano-style guarded types, and clock quantification.

Initial algebras In Section 2 we present a type system that includes a notion of strictly positive type operator. We include a least fixpoint operator $\mu X.F[X]$. For normal strictly positive type operators F (i.e. ones that do not involve the $\triangleright^{\kappa}$ operator), $\mu X.F[X]$ is exactly the normal least fixpoint of F. Thus we can define the normal well-founded types of natural numbers, lists, trees and so on. Our calculus contains constructors $\mathsf{Cons}_F :: F[\mu X.F] \to \mu X.F$ for building values of these types, and recursion combinators for eliminating values:

$$\mathsf{primRec}_{F,A} :: (F[(\mu X.F) \times A] \to A) \to \mu X.F[X] \to A$$

We opt to use primitive recursion instead of a fold operator (i.e. $\mathit{fold} :: (F[A] \to A) \to \mu X.F[X] \to A$) because it allows for constant time access to the top-level of an inductive data structure.

We show in the proof of Theorem 2 that $\mu X.F$ is the carrier of the initial F-algebra in our model. Therefore, we may use standard reasoning techniques about initial algebras to reason about programs written using the primRec combinator.

Guarded final coalgebras When we consider strictly positive type operators of the form $F[\triangleright^{\kappa} X]$, where κ does not appear in F then, in addition to $\mu X.F[\triangleright^{\kappa} X]$ being the least fixpoint of the operator $F[\triangleright^{\kappa} -]$, it is also the *greatest* fixpoint. This initial-final coincidence is familiar from domain theoretic models of recursive types (see, e.g., Smyth and Plotkin [23]), and has previously been observed as a feature of guarded recursive types by Birkedal *et al.* [6].

The *unfold* combinator that witnesses $\mu X.F[\triangleright^{\kappa} X]$ as final can be implemented in terms of the **fix** operator, and Cons:

$$\begin{aligned}&\mathit{unfold}_F :: \forall\kappa.(A \to F[\triangleright^{\kappa} A]) \to A \to \mu X.F[\triangleright^{\kappa} X]\\&\mathit{unfold}_F = \Lambda\kappa.\lambda f.\textbf{fix}\ (\lambda g\ a.\mathsf{Cons}\ (\mathit{fmap}_F\ (\lambda x.g \circledast x)\ (f\ a)))\end{aligned}$$

where $\mathit{fmap}_F :: (A \to B) \to (F[A] \to F[B])$ is the functorial mapping combinator associated with the strictly positive type operator F. Note that without the additional $\triangleright^{\kappa}$ in $F[\triangleright^{\kappa} -]$, there would be no way to define *unfold*, due to the typing of **fix**.

To make observations on elements, we define a deCons_F combinator for every F, using the primitive recursion

$$\begin{aligned}&\mathit{deCons}_F :: \forall\kappa.(\mu X.F[\triangleright^{\kappa} X]) \to F[\triangleright^{\kappa} \mu X.F[\triangleright^{\kappa} X]]\\&\mathit{deCons}_F = \Lambda\kappa.\mathsf{primRec}\ (\lambda x.\ \mathit{fmap}_{F[\triangleright^{\kappa} -]}\ (\lambda y.\textbf{fst}\ y)\ x)\end{aligned}$$

The proof of Theorem 3 shows that these definitions of unfold_F and deCons_F exhibit $\mu X.F[\triangleright^{\kappa} X]$ as the final $F[\triangleright^{\kappa} -]$-coalgebra, using deCons_F as the structure map. This means that $\mathit{unfold}_F[\kappa]f$ is the unique $F[\triangleright^{\kappa} -]$-coalgebra homomorphism from A to $\mu X.F[\triangleright^{\kappa} -]$.

Final coalgebras Theorem 3 only gives us final coalgebras in the case when the recursion in the type is guarded by the type constructor $\triangleright^{\kappa} -$. So we do not yet have a dual to the least fixpoint type constructor $\mu X.F$. However, as we hinted above in the case of streams, we can use clock quantification to construct a final F-coalgebra, where F need not contain an occurrence of $\triangleright^{\kappa} -$.

For the type $\forall\kappa.\mu X.F[\triangleright^{\kappa} -]$ we define an unfold^{ν}_F operator, in a similar style to the unfold_F operator above. However, this operator takes an argument of type $(A \to F[A])$, with no mention of guardedness.

$$\begin{aligned}&\mathit{unfold}^{\nu}_F :: (A \to F[A]) \to A \to \forall\kappa.\mu X.F[\triangleright^{\kappa} X]\\&\mathit{unfold}^{\nu}_F = \lambda f\ a.\Lambda\kappa.\\&\qquad \textbf{fix}\ (\lambda g\ a.\mathsf{Cons}_F\ (\mathit{fmap}_F(\lambda x.g \circledast \textbf{pure}\ x)\ (f\ a)))\ a\end{aligned}$$

We can also define the corresponding deconstructor, building on the definition of deCons_F above, but also using the **force** construct in conjunction with clock quantification. This is a generalisation of the definition of deStreamCons above.

$$\begin{aligned}&\mathit{deCons}^{\nu}_F :: (\forall\kappa.\mu X.F[\triangleright^{\kappa} X]) \to F[\forall\kappa.\mu X.F[\triangleright^{\kappa} X]]\\&\mathit{deCons}^{\nu}_F = \lambda x.\ \mathit{fmap}_F\ (\lambda x.\textbf{force}\ x)\ (\Lambda\kappa.\ \mathit{deCons}_F[\kappa]\ (x[\kappa]))\end{aligned}$$

In the application of fmap_F, we make use of the type equalities in our calculus, which allow us to treat the types $\forall\kappa.F[X]$ and $F[\forall\kappa.X]$ as equivalent, when κ does not appear in F. We present the type equalities in Section 2.2, and prove them sound in Section 3.4.

Our last main technical result, Theorem 4, states that, in the semantics we define Section 3, $\forall\kappa.\mu X.F[\triangleright^{\kappa} -]$ actually is the final F-coalgebra. The use of clock quantification has allowed us to

remove mention of the guardedness type constructor, and given us a way to safely program and reason about F-coalgebras.

1.5 Circular traversals of trees

We now present a perhaps unexpected application of the use of clock quantification that does not relate to ensuring productivity of recursive programs generating infinite data. An interesting use of laziness in functional programming languages is to perform single-pass transformations of data structures that would appear to require at least two passes. A classic example, due to Bird [3], is the *replaceMin* function that replaces each value in a binary tree with the minimum of all the values in the tree, in a single pass. In normal Haskell, this function is written as follows:

```
replaceMin :: Tree → Tree
replaceMin t = let (t′, m) = replaceMinBody t m in t′
  where
    replaceMinBody (Leaf x) m = (Leaf m, x)
    replaceMinBody (Br l r) m =
        let (l′, m_l) = replaceMinBody l m
            (r′, m_r) = replaceMinBody r m
        in (Br l′ r′, min m_l m_r)
```

The interesting part of this function is the first **let** expression. This takes the minimum value m computed by the traversal of the tree and passes it into the *same* traversal to build the new tree. This function is a marvel of declarative programming: we have declared that we wish the tree t' to be labelled with the minimum value in the tree t just by stating that they be the same, without explaining at all why this definition makes any sense. Intuitively, the reason that this works is that the overall minimum value is never used in the computation of the minimum, only the construction of the new tree. The computation of the minimum and the construction of the new tree conceptually exist at different moments in time, and it is only safe to treat them the same after both have finished.

Using explicit clock variables we can give a version of the *replaceMin* definition that demonstrates that we have actually defined a total function from trees to trees. The use of clock variables allows us to be explicit about the time when various operations are taking place.

Out first step is to replace the circular use of **let** with a *feedback* combinator defined in terms of the **fix** operator:

```
feedback : ∀κ. (▷^κ U → (B[κ], U)) → B[κ]
feedback = Λκ.λf.fst (fix (λx. f (pure (λx.snd x) ⊛ x)))
```

In the application of the **fix** operator, $x : \triangleright^\kappa(B \times U)$. The notation $B[\kappa]$ indicates that κ may appear free in the type B.

We now rewrite the *replaceMinBody* function so that we can apply *feedback* to it. We have highlighted the changes from the previous definition.

```
replaceMinBody :: Tree → ∀κ. ▷^κ Integer → (▷^κ Tree, Integer)
replaceMinBody (Leaf x) m = (pure Leaf ⊛ m, x)
replaceMinBody (Br l r) m =
        let (l′, m_l) = replaceMinBody l m
            (r′, m_r) = replaceMinBody r m
        in (pure Br ⊛ l′ ⊛ r′, min m_l m_r)
```

The changes required are minor: we must change the type, and use the applicative functor structure of $\triangleright^\kappa$ to indicate when computations are taking place "tomorrow".

Applying *feedback* to *replaceMinBody*, and using **force** to remove the now redundant occurrence of $\triangleright^\kappa$, we can define the full *replaceMin* function in our system:

```
replaceMin :: Tree → Tree
replaceMin t = force (Λκ.feedback[κ] (replaceMinBody[κ]) t)
```

$$\frac{X \in \Theta}{\Delta;\Theta \vdash X : \mathtt{type}} \qquad \frac{}{\Delta;\Theta \vdash 1 : \mathtt{type}}$$

$$\frac{\Delta;\Theta \vdash A : \mathtt{type} \qquad \Delta;\Theta \vdash B : \mathtt{type}}{\Delta;\Theta \vdash A \times B : \mathtt{type}}$$

$$\frac{\Delta;\Theta \vdash A : \mathtt{type} \qquad \Delta;\Theta \vdash B : \mathtt{type}}{\Delta;\Theta \vdash A + B : \mathtt{type}}$$

$$\frac{\Delta;- \vdash A : \mathtt{type} \qquad \Delta;\Theta \vdash B : \mathtt{type}}{\Delta;\Theta \vdash A \to B : \mathtt{type}}$$

$$\frac{\Delta;\Theta, X \vdash A : \mathtt{type}}{\Delta;\Theta \vdash \mu X.A : \mathtt{type}} \qquad \frac{\Delta,\kappa;\Theta \vdash A : \mathtt{type}}{\Delta;\Theta \vdash \forall\kappa.A : \mathtt{type}}$$

$$\frac{\Delta;\Theta \vdash A : \mathtt{type} \qquad \kappa \in \Delta}{\Delta;\Theta \vdash \triangleright^\kappa A : \mathtt{type}}$$

Figure 1. Well-formed types and type operators

By the soundness property of our system that we prove in Section 3, we are assured that we have defined a total function from trees to trees. The standard Haskell type system does not provide this guarantee.

1.6 Models of guarded recursion

To substantiate the claims we have made in this introduction, in Section 3 we construct a multiply-step-indexed model of the type system we present in the next section. The multiple step-indexes are required for the multiple clock variables in our system, each representing separate time streams. Step-indexed models were introduced by Appel and McAllester [2] to prove properties of recursive types. The use of step-indexing serves to break the circularity inherent in recursive types. Dreyer *et al.* [10] exposed the connection between Nakano's guarded recursion and step indexed models. More recently, Birkedal *et al.* [4, 6] have elaborated this connection, particularly in the direction of dependent types.

Alternative semantics for guarded recursion have involved ultrametric spaces, for example Birkedal *et al.* [5] and Krishnaswami and Benton [15]. Hutton and Jaskelioff [13] have also used an approach based on metric spaces for identifying productive stream generating functions.

Finally, we mention the syntactic approach of Severi and de Vries [22], who define the notion of infinitary strong normalisation to prove productivity of dependent type systems with guarded recursion.

2. A type system with clocks and guards

In the introduction, we presented our motivating examples in terms of a Haskell-like language informally extended with a Nakano-style guard modality and clock quantification. To formally state the properties of our combination of clock quantification, clock-indexed guard modalities and inductive types, in this section we define an extension of the simply-typed λ-calculus with these features. In the next section, we define a semantics for our system in which we can formally state the results that we claimed in the introduction.

2.1 Well-formed types with clock variables

The types of our system include two kinds of variable that may occur free and bound: clock variables in the clock quantifier and the clock-indexed guard modality, as well as type variables occurring

in strictly positive positions for the inductive types. We therefore formally define when a type is well-formed with respect to contexts of clock and type variables, using the rules displayed in Figure 1.

The well-formedness judgement for types ($\Delta; \Theta \vdash A : \texttt{type}$) is defined with respect to a *clock context* Δ and a *type variable context* Θ. Clock contexts are lists of clock variables, $\Delta = \kappa_1, ..., \kappa_n$, and type variable contexts are lists of type variable names, $\Theta = X_1, ..., X_n$. Type variables are only used to manage the scope of the nested least fixpoint type operator μ; there is no type polymorphism in the type system we present here. We generally use Roman letters A, B, C to stand for types, but we also occasionally use F and G when we wish to emphasise that types with free type variables are strictly positive type operators. For any type A, we define $fc(A)$ to be the set of free clock variables that appear in A.

Most of the type well-formedness rules are standard: we have a rule for forming a type from a type variable that is in scope and rules for forming the unit type 1, product types $A \times B$, sum types $A + B$, function types $A \to B$ and least fixpoint types $\mu X.A$. We enforce the restriction to strictly positive type operators by disallowing occurrences of free type variables in the domain component of function types. In Section 2.3, we will see that each of the standard type constructors will have the usual corresponding term-level constructs; our system subsumes the simply typed λ-calculus with products, sums and least fixpoint types.

The remaining two rules are for forming clock quantification $\forall\kappa.A$ types and the clock-indexed Nakano-style guard modality $\triangleright^\kappa A$. The type formation rule for clock quantification is very similar to universal type quantification from standard polymorphic type theories like System F.

As we described in the introduction in Section 1.3, we have augmented the Nakano-style delay modality with a clock variable. This allows us to distinguish between guardedness with respect to multiple clocks, and hence to be able to close over all the steps guarded by a particular clock.

2.2 Type Equality

The examples we presented in the introduction made use of several type equalities allowing us to move clock quantification through types. For instance, in the definition of the deStreamCons stream deconstructor in Section 1.3, we made use of a type equality to treat a value of type $\forall\kappa$.Integer as having the type Integer. Intuitively, this is valid because a value of type Integer exists independently of any clock, therefore we can remove the clock quantification. In general, clock quantification commutes with almost all strictly positive type formers, except for guards indexed by the same clock variable. For those, we must use **force**.

The following rules are intended to be read as defining a judgement $\Delta; \Theta \vdash A \equiv B : \texttt{type}$, indicating that the well-formed types $\Delta; \Theta \vdash A : \texttt{type}$ and $\Delta; \Theta \vdash B : \texttt{type}$ are to be considered equal. In addition to these rules, the relation $\Delta \vdash A \equiv B : \texttt{type}$ is reflexive, symmetric, transitive and a congruence.

$$\begin{array}{rl}
\forall\kappa.A \equiv A & (\kappa \notin fc(A)) \\
\forall\kappa.A + B \equiv (\forall\kappa.A) + (\forall\kappa.B) & \\
\forall\kappa.A \times B \equiv (\forall\kappa.A) \times (\forall\kappa.B) & \\
\forall\kappa.A \to B \equiv A \to \forall\kappa.B & (\kappa \notin fc(A)) \\
\forall\kappa.\forall\kappa'.A \equiv \forall\kappa'.\forall\kappa.A & (\kappa \neq \kappa') \\
\forall\kappa.\triangleright^{\kappa'} A \equiv \triangleright^{\kappa'} \forall\kappa.A & (\kappa \neq \kappa') \\
\forall\kappa.\mu X.F[A, X] \equiv \mu X.F[\forall\kappa.A, X] & (fc(F) = \emptyset)
\end{array}$$

In the last rule, F must be strictly positive in A as well as X.

After we define the terms of our system in the next sub-section, it will become apparent that several of the type equality rules could be removed, and their use replaced by terms witnessing the conversions back and forth. In the case of product types, we could define the following pair of terms (using the syntax and typing rules we define below):

$$\lambda x.(\Lambda\kappa.\mathbf{fst}\ (x[\kappa]), \Lambda\kappa.\mathbf{snd}\ (x[\kappa])) \\ : (\forall\kappa.A \times B) \to (\forall\kappa.A) \times (\forall\kappa.B)$$

and

$$\lambda x.\Lambda\kappa.(\mathbf{fst}\ x\ [\kappa], \mathbf{snd}\ x\ [\kappa]) \\ : (\forall\kappa.A) \times (\forall\kappa.B) \to \forall\kappa.A \times B$$

Indeed, the denotational semantics we present in Section 3 will interpret both of these functions as the identity. However, not all the type equality rules are expressible in terms of clock abstraction and application, for instance, the $\forall\kappa.A \equiv A$ rule and the rule for sum types. Moreover, our definition of deCons in Section 1.4 is greatly simplified by having clock quantification commute with all type formers uniformly (recall that we made crucial use of the equality $\forall\kappa.F[-] \equiv F[\forall\kappa.-]$). Therefore, we elect to include type equalities for commuting clock quantification with all type formers uniformly.

2.3 Well-typed Terms

The terms of our system are defined by the following grammar:

$$\begin{array}{rl}
e, f, g ::= & x \mid * \mid \lambda x.\ e \mid f e \mid (e_1, e_2) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \\
& \mid \mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{case}\ e\ \mathbf{of\ inl}\ x.f; \mathbf{inr}\ y.g \\
& \mid \mathsf{Cons}_F\ e \mid \mathsf{primRec}_F\ e \mid \Lambda\kappa.e \mid e[\kappa'] \\
& \mid \mathbf{pure}\ x \mid f \circledast e \mid \mathbf{fix}\ f \mid \mathbf{force}\ e
\end{array}$$

The *well-typed* terms of our system are defined by the typing judgement $\Delta; \Gamma \vdash e : A$, given by the rules presented in Figure 2. Terms are judged to be well-typed with respect to a clock variable context Δ, and a typing context Γ consisting of a list of variable : type pairs: $x_1 : A_1, ..., x_n : A_n$. We only consider typing contexts where each type is well-formed with respect to the clock variable context Δ, and the *empty* type variable context. The result type A in the typing judgement must also be well-formed with respect to Δ and the empty type variable context.

We have split the typing rules in Figure 2 into five groups. The first group includes the standard typing rules for the simply-typed λ-calculus with unit, product and sum types. Apart from the additional clock variable contexts Δ, these rules are entirely standard. The second group contains the rules for the inductive types $\mu X.F$. Again, apart from the appearance of clock variable contexts, these rules are the standard constructor and primitive recursion constructs for inductive types.

The third group of rules cover clock abstraction and application, and the integration of the type equality rules we defined in the previous section. In the κ-App rule, we have used the notation $A[\kappa \mapsto \kappa']$ to indicate the substitution of the clock variable κ' for κ. These rules are as one might expect for System F-style quantification with non-trivial type equality, albeit that in the κ-App rule the κ' clock variable cannot already appear in the type. This disallows us from using the same clock variable twice, preventing multiple "time-streams" becoming conflated.

The fourth group of rules state that the clock-indexed guard modality supports the **pure** and apply ($\circledast$) operations of an applicative functor. When we define a denotational semantics of our calculus in the next section, these operations will be interpreted simply as the identity function and normal function application respectively, meaning that the applicative functor laws hold trivially.

Finally, the fifth group of rules presents the typing for Nakano's guarded **fix** combinator, now indexed by a clock variable, and our **force** combinator for removing the guard modality when it is protected by a clock quantification.

2.4 Type operators are functorial

In Section 1.4 we used a functorial mapping function $fmap_F : (A \to B) \to F[A] \to F[B]$, which we claimed was defined

1. Simply-typed λ-calculus with products and sums

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{}{\Delta; \Gamma \vdash * : 1}\ (\textsc{Unit}) \qquad \frac{\Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x.\, e : A \to B}\ (\textsc{Abs}) \qquad \frac{\Delta; \Gamma \vdash f : A \to B \qquad \Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash f e : B}\ (\textsc{App})$$

$$\frac{\Delta; \Gamma \vdash e_1 : A \qquad \Delta; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash (e_1, e_2) : A \times B}\ (\textsc{Pair}) \qquad \frac{\Delta; \Gamma \vdash e : A \times B}{\Delta; \Gamma \vdash \mathbf{fst}\ e : A}\ (\textsc{Fst}) \qquad \frac{\Delta; \Gamma \vdash e : A \times B}{\Delta; \Gamma \vdash \mathbf{snd}\ e : B}\ (\textsc{Snd}) \qquad \frac{\Delta; \Gamma \vdash e : A}{\Delta; \Gamma \vdash \mathbf{inl}\ e : A + B}\ (\textsc{Inl})$$

$$\frac{\Delta; \Gamma \vdash e : B}{\Delta; \Gamma \vdash \mathbf{inr}\ e : A + B}\ (\textsc{Inr}) \qquad \frac{\Delta; \Gamma \vdash e : A + B \qquad \Delta; \Gamma, x : A \vdash f : C \qquad \Delta; \Gamma, y : B \vdash g : C}{\Delta; \Gamma \vdash \mathbf{case}\ e\ \mathbf{of\ inl}\ x.f; \mathbf{inr}\ y.g : C}\ (\textsc{Case})$$

2. Least Fixpoint Types

$$\frac{\Delta; \Gamma \vdash e : F[\mu X.F[X]]}{\Delta; \Gamma \vdash \mathsf{Cons}_F\ e : \mu X.F[X]}\ (\textsc{Cons}) \qquad \frac{\Delta; \Gamma \vdash e : F[(\mu X.F[X]) \times A] \to A}{\Delta; \Gamma \vdash \mathsf{primRec}_F\ e : \mu X.F[X] \to A}\ (\textsc{PrimRec})$$

3. Clock Abstraction and Application, and Type Equality

$$\frac{\Delta, \kappa; \Gamma \vdash e : A \qquad \kappa \notin fc(\Gamma)}{\Delta; \Gamma \vdash \Lambda\kappa.e : \forall\kappa.A}\ (\kappa\text{-}\textsc{Abs}) \qquad \frac{\Delta; \Gamma \vdash e : \forall\kappa.A \qquad \kappa' \in \Delta \qquad \kappa' \notin fc(A)}{\Delta; \Gamma \vdash e[\kappa'] : A[\kappa \mapsto \kappa']}\ (\kappa\text{-}\textsc{App})$$

$$\frac{\Delta; \Gamma \vdash e : A \qquad \Delta \vdash A \equiv B}{\Delta; \Gamma \vdash e : B}\ (\textsc{TyEq})$$

4. Applicative Functor Structure for $\triangleright^\kappa -$

$$\frac{\Delta; \Gamma \vdash e : A \qquad \kappa \in \Delta}{\Delta; \Gamma \vdash \mathbf{pure}\ e : \triangleright^\kappa A}\ (\textsc{DePure}) \qquad \frac{\Delta; \Gamma \vdash f : \triangleright^\kappa (A \to B) \qquad \Delta; \Gamma \vdash e : \triangleright^\kappa A}{\Delta; \Gamma \vdash f \circledast e : \triangleright^\kappa B}\ (\textsc{DeApp})$$

5. Fix and Force

$$\frac{\Delta; \Gamma \vdash f : \triangleright^\kappa A \to A}{\Delta; \Gamma \vdash \mathbf{fix}\ f : A}\ (\textsc{Fix}) \qquad \frac{\Delta; \Gamma \vdash e : \forall\kappa.\triangleright^\kappa A}{\Delta; \Gamma \vdash \mathbf{force}\ e : \forall\kappa.A}\ (\textsc{Force})$$

Figure 2. Well-typed terms

$$\begin{aligned}
&\mathit{fmap}_F : \overrightarrow{(A \to B)} \to F[\vec{A}] \to F[\vec{B}] \\
&\mathit{fmap}_{X_i}\ \vec{f} x = f_i\ x \\
&\mathit{fmap}_1\ \vec{f} x = * \\
&\mathit{fmap}_{F \times G}\ \vec{f} x = (\mathit{fmap}_F\ \vec{f}\ (\mathbf{fst}\ x), \mathit{fmap}_G\ \vec{f}\ (\mathbf{snd}\ x)) \\
&\mathit{fmap}_{F+G}\ \vec{f} x = \mathbf{case}\ x\ \mathbf{of\ inl}\ y.\mathbf{inl}\ (\mathit{fmap}_F\ \vec{f}\ y) \\
&\qquad\qquad\qquad\qquad \mathbf{inr}\ z.\mathbf{inr}\ (\mathit{fmap}_G\ \vec{f}\ z) \\
&\mathit{fmap}_{A \to F}\ \vec{f} x = \lambda a.\, \mathit{fmap}_F\ \vec{f}\ (x\ a) \\
&\mathit{fmap}_{\triangleright^\kappa F}\ \vec{f} x = \mathbf{pure}\ (\mathit{fmap}_F\ \vec{f}) \circledast x \\
&\mathit{fmap}_{\forall\kappa.F}\ \vec{f} x = \Lambda\kappa.\mathit{fmap}_F\ \vec{f}\ (x[\kappa]) \\
&\mathit{fmap}_{\mu X.F} \vec{f} x = \mathsf{primRec}\ (\lambda x.\mathsf{Cons}(\mathit{fmap}_F\ \vec{f}\ (\lambda x.\ \mathbf{snd}\ x)\ x))\ x
\end{aligned}$$

Figure 3. fmap_F for all type operators F

for each strictly positive type operator F. We define this operator by structural recursion on the derivation of F, using the clauses in Figure 3. Due to the presence of nested least fixpoint types in our calculus, we handle n-ary strictly positive type operators.

2.5 Programming with guarded types and clocks

Lists and Trees, Finite and Infinite Using the least fixpoint type operator $\mu X.F$, we can reconstruct many of the usual inductive data types used in functional programming. For example, the OCaml declaration:

$$\mathbf{type}\ \text{list} = \mathsf{NilList} \mid \mathsf{ConsList}\ \mathbf{of}\ A \times \text{list}$$

of finite lists of values of type A, can be expressed as the type $\mu X.1 + A \times X$ in our system, where we have replaced the two constructors NilList and ConsList with a use of the sum type former. Likewise, the type of binary trees labelled with As that we used in Section 1.5 can be written as $\mu X.A + X \times X$.

A similar data type declaration in Haskell has a different interpretation. If we make the following declaration in Haskell:

$$\mathbf{data}\ \text{List} = \mathsf{NilList} \mid \mathsf{ConsList}\ A\ \text{List}$$

then the type List includes both finite lists of As, and infinite lists of As (a similar effect can be achieved in OCaml by use of the **lazy** type constructor). Haskell's type system is too weak to distinguish the definitely finite from the possibly infinite case, and it is often left to the documentation to warn the programmer that some functions will be non-productive on infinite lists (for example, the *reverse* function).

Making use of Nakano's guard modality $\triangleright^\kappa -$, we are able to express Haskell-style possibly infinite lists as the guarded inductive type $\mu X.1 + A \times \triangleright^\kappa A$. As we saw in the introduction, infinite lists can be constructed using the guarded **fix** operator. For example, the infinite repetition of a fixed value can be written in our system as:

$$\lambda a.\mathbf{fix}\ (\lambda l.\mathsf{Cons}\ (\mathbf{inr}\ (a, l)))$$

If we restrict ourselves to only one clock variable, and always use guarded recursive types, then we obtain a programming language similar to a monomorphic Haskell, except with a guarantee that all functions are productive.

Co-inductive Streams As we saw in Section 1.2, programming with guarded types is productive, but they are fundamentally incompatible with normal inductive types. Recall that if we define infinite streams of As as $\text{Stream}^\kappa\ A = \mu X.A \times \triangleright^\kappa X$, then there is no way of defining a function $\mathit{tail} : \text{Stream}^\kappa\ A \to \text{Stream}^\kappa\ A$; the best we can do is $\mathit{tail} : \text{Stream}^\kappa\ A \to \triangleright^\kappa \text{Stream}^\kappa\ A$. The rest of the stream only exists "tomorrow".

Clock quantification fixes this problem. We define:

$$\text{Stream}\ A = \forall\kappa.\mu X.A \times \triangleright^\kappa A$$

Now we can define the two deconstructors for making observations on streams, with the right types:

$$\begin{aligned}&\mathit{head}\ :\ \text{Stream}\ A \to A\\ &\mathit{head} = \lambda s.\Lambda\kappa.\mathsf{primRec}\ (\lambda x.\mathbf{fst}\ x)\ (s[\kappa])\end{aligned}$$

$$\begin{aligned}&\mathit{tail}\ :\ \text{Stream}\ A \to \text{Stream}\ A\\ &\mathit{tail} = \lambda s.(\mathbf{force}\ (\Lambda\kappa.\mathsf{primRec}\ (\lambda x.\mathbf{pure}(\lambda x.\mathbf{fst}\ x) \circledast (\mathbf{snd}\ x))\\ &\qquad\qquad (s[\kappa])))\end{aligned}$$

In the definition of *head* we use the fact that κ cannot appear in A to apply the first type equality rule from Section 2.2.

Stream Processing Ghani, Hancock and Pattinson [11] define a type of representations of continuous functions on streams using a least fixpoint type nested within a greatest fixpoint. A continuous function on streams is a function that only requires a finite amount of input for each element of the output. Ghani *et al.*'s type is expressed in our system as follows:

$$\text{SP}\ I\ O = \forall\kappa.\mu X.\mu Y.(I \to Y) + (O \times \triangleright^\kappa X)$$

where SP stands for "Stream Processor" and I and O stand for input and output respectively. In this type, the outer fixpoint permits the production of infinitely many Os, due to the guarded occurrence of X, while the inner fixpoint only permits a finite amount of Is to be read from the input stream. This kind of nested fixpoint is not expressible within Coq, but is possible with Agda's support for coinductive types [9].

Defining the application of an SP $I\ O$ to a stream of Is serves as an interesting example of programming with guards and clocks. We make use of a helper function for unfolding values of type SP $I\ O$ that have already been instantiated with some clock:

$$\mathit{step} : \mathit{SP}^\kappa\ I\ O \to \mu Y.(I \to Y) \times (O \times \triangleright^\kappa(\mathit{SP}^\kappa\ I\ O))$$

where $\mathit{SP}^\kappa\ I\ O$ is $\mathit{SP}\ I\ O$ instantiated with the clock κ.

The definition of stream processor application goes as follows, where we permit ourselves a little pattern matching syntactic sugar for pairs:

$$\begin{aligned}&\mathit{apply} : \text{SP}\ I\ O \to \text{Stream}\ I \to \text{Stream}\ O\\ &\mathit{apply} =\\ &\lambda sp\ s.\Lambda\kappa.\mathbf{fix}(\lambda rec\ sp\ s.\\ &\qquad \mathsf{primRec}\ (\lambda x\ s.\mathbf{case}\ x\ \mathbf{of}\\ &\qquad\qquad\qquad \mathbf{inl}\ f.\ f\ (\mathit{head}\ s)(\mathit{tail}\ s)\\ &\qquad\qquad\qquad \mathbf{inr}\ (o, sp').\\ &\qquad\qquad\qquad\quad \mathsf{Cons}(o, rec \circledast sp' \circledast \mathbf{pure}\ s))\\ &\qquad\quad (\mathit{step}\ sp))\\ &\qquad (sp\ [\kappa])\ s\end{aligned}$$

This function consists of two nested loops. The outer loop, expressed using **fix**, generates the output stream. This loop is running on the clock κ, introduced by the $\Lambda\kappa$. We have instantiated the stream processor with the same clock; this causes the steps of the output stream to be in lockstep with the output steps of the stream processor, exactly as we might expect. The inner loop, expressed using primRec, recurses over the finite list of input instructions from the stream processor, pulling items from the input stream as needed. The input stream is *not* instantiated with the same clock as the stream processor and the output stream: we need to access an arbitrary number of elements from the input stream for each step of the stream processor, so their clocks cannot run in lockstep.

The Partiality Monad The partiality monad is a well-known coinductively defined monad that allows the definition of functions via general recursion, even in a total language. The partiality monad is defined as $\text{Partial}\ A = \nu X.A + X$. Hence, a value of type $\text{Partial}\ A$ consists of a stream of **inr**s, either continuing forever, or terminating in an **inl** with a value of type A. Using the partiality monad, possibly non-terminating computations can be represented, with non-termination represented by an infinite stream that never returns a value. Capretta [7] gives a detailed description.

To express the partiality monad in our system, we decompose it into a guarded least fixpoint and clock quantification, just as we did for streams and stream processors above:

$$\begin{aligned}\text{Partial}^\kappa\ A &= \mu X.A + \triangleright^\kappa X\\ \text{Partial}\ A &= \forall\kappa.\text{Partial}^\kappa\ A\end{aligned}$$

For convenience, we define two constructors for $\text{Partial}^\kappa\ A$, corresponding to the two components of the sum type:

$$\begin{aligned}&\mathit{now}^\kappa : A \to \text{Partial}^\kappa\ A\\ &\mathit{now}^\kappa = \lambda a.\ \mathsf{Cons}\ (\mathbf{inl}\ a)\end{aligned}$$

$$\begin{aligned}&\mathit{later}^\kappa : \triangleright^\kappa(\text{Partial}^\kappa\ A) \to \text{Partial}^\kappa\ A\\ &\mathit{later}^\kappa = \lambda p.\ \mathsf{Cons}\ (\mathbf{inr}\ p)\end{aligned}$$

It is straightforward to use guarded recursion and clock quantification to define the *return* and bind functions that demonstrate that Partial is indeed a monad. For an example of a possibly non-terminating function that can be expressed using the partiality monad, we define the following function *collatz*. For each natural number n, this function only terminates if the Collatz sequence starting from n reaches 1. Whether or not this sequence reaches 1 for all n is a famous unsolved question in number theory.

$$\begin{aligned}&\mathit{collatz} : \text{Natural} \to \text{Partial}\ 1\\ &\mathit{collatz} = \lambda n.\Lambda\kappa.\mathbf{fix}\ (\lambda rec\ n.\mathbf{if}\ n = 1\ \mathbf{then}\ \mathit{now}^\kappa\ (*)\\ &\qquad\qquad\qquad\qquad \mathbf{else\ if}\ n\ \mathit{mod}\ 2 = 0\ \mathbf{then}\\ &\qquad\qquad\qquad\qquad\quad \mathit{later}^\kappa\ (rec \circledast (\mathbf{pure}\ (n/2)))\\ &\qquad\qquad\qquad\qquad \mathbf{else}\\ &\qquad\qquad\qquad\qquad\quad \mathit{later}^\kappa\ (rec \circledast (\mathbf{pure}\ (3 * n + 1)))))\ n\end{aligned}$$

The partiality monad is not a drop-in replacement for general recursion. The type Partial A reveals information about the number of steps that it takes to compute a value of type A. Therefore, it is possible to write a timeout function that runs a partial computation for a fixed number of steps before giving up:

$$\mathit{timeout} : \text{Natural} \to \text{Partial}\ A \to A + 1$$

The partiality monad allows us to write possibly non-terminating functions, but also allows us to make more observations on them.

3. A denotational semantics for clocks and guards

We have claimed that the typing discipline from the previous section guarantees that programs will be productive. We now substantiate this claim by defining a domain-theoretic model of our system, and showing that the denotation of a well-typed closed term is never $\bot$. We accomplish this by using a multiply-step-indexed interpretation of types, where the multiple step indexes correspond to the multiple clock variables that may be in scope.

$$
\begin{array}{rcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![*]\!]\eta & = & \mathsf{Unit} \\
[\![\lambda x.\, e]\!]\eta & = & \mathsf{Lam}\ (\lambda v.\, [\![e]\!](\eta[x \mapsto v])) \\
[\![f e]\!]\eta & = & \begin{cases} d_f\ ([\![e]\!]\eta) & \text{if } [\![f]\!]\eta = \mathsf{Lam}\ d_f \\ \bot & \text{otherwise} \end{cases} \\
[\![(e_1, e_2)]\!]\eta & = & \mathsf{Pair}\ ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![\mathbf{fst}\ e]\!]\eta & = & \begin{cases} d_1 & \text{if } [\![e]\!]\eta = \mathsf{Pair}\ (d_1, d_2) \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathbf{snd}\ e]\!]\eta & = & \begin{cases} d_2 & \text{if } [\![e]\!]\eta = \mathsf{Pair}\ (d_1, d_2) \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathbf{inl}\ e]\!]\eta & = & \mathsf{Inl}\ ([\![e]\!]\eta) \\
[\![\mathbf{inr}\ e]\!]\eta & = & \mathsf{Inr}\ ([\![e]\!]\eta) \\
\left[\!\!\left[\begin{array}{l} \mathbf{case}\ e\ \mathbf{of} \\ \quad \mathbf{inl}\ x.f \\ \quad \mathbf{inr}\ y.g \end{array} \right]\!\!\right] \eta & = & \begin{cases} [\![f]\!](\eta[x \mapsto d]) & \text{if } [\![e]\!]\eta = \mathsf{Inl}\ d \\ [\![g]\!](\eta[y \mapsto d]) & \text{if } [\![e]\!]\eta = \mathsf{Inr}\ d \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4. Semantics for the simply-typed portion of our system

3.1 Semantics of terms

We interpret terms in (a mild extension of) the standard domain theoretic model of the untyped lazy λ-calculus, described by Pitts [21]. A feature of this semantics is the erasure of anything to do with the guardedness type constructor $\triangleright^\kappa -$ and the clock quantification. The **fix** operator is interpreted as the domain-theoretic fixpoint, i.e., exactly the usual interpretation of general recursion.

We assume a directed complete partial order with bottom ($\mathrm{DCPO}_\bot$), D, satisfying the following recursive domain equation:

$$D \cong (D \to D)_\bot \oplus (D \times D)_\bot \oplus D_\bot \oplus D_\bot \oplus 1_\bot$$

where $\oplus$ represents the coalesced sum that identifies the $\bot$ element of all the components. We are guaranteed to be able to obtain such a D by standard results about the category $\mathrm{DCPO}_\bot$ (see, e.g., Smyth and Plotkin [23]). The five components of the right hand side of this equation will be used to carry functions, products, the left and right injections for sum types and the unit value respectively. We use the following symbols to represent the corresponding continuous injective maps into D:

$$
\begin{array}{ll}
\mathsf{Lam} : (D \to D) \to D & \mathsf{Pair} : D \times D \to D \\
\mathsf{Inl} : D \to D & \mathsf{Inr} : D \to D \\
\mathsf{Unit} : 1 \to D &
\end{array}
$$

We will write Unit instead of $\mathsf{Unit}\ *$.

Let V be the set of all possible term-level variable names. Environments η are modelled as maps from V to elements of D. Terms are interpreted as continuous maps from environments to elements of D. The clauses for defining the interpretation of parts of our system that are just the simply-typed λ-calculus are standard, and displayed in Figure 4. It can be easily checked that this collection of equations defines a continuous function $[\![-]\!] : (V \to D) \to D$, since everything is built from standard parts.

The applicative functor structure for the guard modality is interpreted just as the identity function and normal application, as we promised in Section 2.3:

$$
\begin{array}{rcl}
[\![\mathbf{pure}\ e]\!]\eta & = & [\![e]\!]\eta \\
[\![f \circledast e]\!]\eta & = & \begin{cases} d_f\ ([\![e]\!]\eta) & \text{if } [\![f]\!]\eta = \mathsf{Lam}\ d_f \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Our interpretation simply translates the **fix** operator as the usual fixpoint operator $\mathrm{fix}\ f = \bigsqcup_n f^n(\bot)$ in $\mathrm{DCPO}_\bot$. The **force** operator is interpreted as a no-operation.

$$
\begin{array}{rcl}
[\![\mathbf{fix}\ f]\!]\eta & = & \begin{cases} \mathrm{fix}\ d_f & \text{if } [\![f]\!]\eta = \mathsf{Lam}\ d_f \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathbf{force}\ e]\!]\eta & = & [\![e]\!]\eta
\end{array}
$$

Finally, we interpret the constructor Cons_F and primitive recursion eliminator $\mathsf{primRec}_F$ for least fixpoint types. The interpretation of construction is straightforward: the constructor itself disappears at runtime, so the interpretation simply ignores it:

$$[\![\mathsf{Cons}_F\ e]\!]\eta \quad = \quad [\![e]\!]\eta$$

To define the semantic counterpart of the $\mathsf{primRec}_F$ operator, we first define a higher-order continuous function $\mathrm{primrec} : (D \to D \to D) \to (D \to D) \to D$ in the model. This is very similar to how one would define a generic $\mathsf{primRec}_F$ in Haskell using general recursion, albeit here in an untyped fashion.

$$
\begin{array}{l}
\mathrm{primrec}\ \mathit{fmap}\ f = \\
\quad \mathsf{Lam}\ (\mathrm{fix}\ (\lambda g x.\ f\ (\mathit{fmap}\ (\mathsf{Lam}\ (\lambda x.\mathsf{Pair}\ (x, g\ x))))\ x))
\end{array}
$$

The first argument to primrec is intended to specify a functorial mapping function. For each of the syntactic types $\Delta; \Theta \vdash F : \mathtt{type}$ defined in Figure 1, we define a suitable $\mathrm{fmap}_\mathrm{F} : D^{|\Theta|} \to D \to D$, where $|\Theta|$ is the number of type variables in Θ. This definition is presented in Figure 5.

With these definitions we can define the interpretation of the syntactic $\mathsf{primRec}_F$ operator. Note that the syntactic type constructor F here always has exactly one free type variable, so fmap_F has type $D \to D \to D$, as required by primrec.

$$
\begin{array}{l}
[\![\mathsf{primRec}_F\ f]\!]\eta = \\
\quad \begin{cases} \mathrm{primrec}\ \mathrm{fmap}_\mathrm{F}\ d_f & \text{if } [\![f]\!]\eta = \mathsf{Lam}\ d_f \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

This completes the interpretation of the terms of our programming language in our untyped model. We now turn to the semantic meaning of types, with the aim of showing, in Section 3.5, that each well-typed term's interpretation is semantically well-typed.

$$
\begin{array}{rcl}
\mathrm{fmap}_\mathrm{X}\ \vec{f}\ x & = & d_f(x) \quad (f_X = \mathsf{Lam}\ d_f) \\
\mathrm{fmap}_1\ \vec{f}\ x & = & \mathsf{Unit} \\
\mathrm{fmap}_{\mathrm{F}\times\mathrm{G}}\ \vec{f}\ (\mathsf{Pair}\ (x, y)) & = & \mathsf{Pair}\ (\mathrm{fmap}_\mathrm{F}\ \vec{f}\ x, \mathrm{fmap}_\mathrm{G}\ \vec{f}\ y) \\
\mathrm{fmap}_{\mathrm{F}+\mathrm{G}}\ \vec{f}\ (\mathsf{Inl}\ x) & = & \mathsf{Inl}\ (\mathrm{fmap}_\mathrm{F}\ \vec{f}\ x) \\
\mathrm{fmap}_{\mathrm{F}+\mathrm{G}}\ \vec{f}\ (\mathsf{Inr}\ x) & = & \mathsf{Inr}\ (\mathrm{fmap}_\mathrm{G}\ \vec{f}\ x) \\
\mathrm{fmap}_{\mathrm{A}\to\mathrm{F}}\ \vec{f}\ (\mathsf{Lam}\ g) & = & \mathsf{Lam}\ (\lambda v.\mathrm{fmap}_\mathrm{F}\ \vec{f}\ (g\ v)) \\
\mathrm{fmap}_{\triangleright^\kappa \mathrm{F}}\ \vec{f}\ x & = & \mathrm{fmap}_\mathrm{F}\ \vec{f}\ x \\
\mathrm{fmap}_{\forall\kappa.\mathrm{F}}\ \vec{f}\ x & = & \mathrm{fmap}_\mathrm{F}\ \vec{f}\ x \\
\mathrm{fmap}_{\mu\mathrm{X}.\mathrm{F}}\ \vec{f}\ x & = & \mathrm{primrec}\ (\mathrm{fmap}_\mathrm{F}\ \vec{f}) \\
 & & \qquad (\mathrm{fmap}_\mathrm{F}\ \vec{f}\ \mathrm{snd})\ x
\end{array}
$$

where

$$\mathrm{snd}\ (\mathsf{Pair}\ (x, y)) = y$$

All unhandled cases are defined to be equal to $\bot$.

Figure 5. Semantic fmap_F for $\Delta; \Theta \vdash F : \mathtt{type}$

3.2 Interpretation of clock variables

For a clock context Δ, we define $[\![\Delta]\!]$ to be the set of clock environments: mappings from the clock variables in Δ to the natural numbers. We use δ, δ' etc. to range over clock environments. For δ and δ' in $[\![\Delta]\!]$, we say that $\delta \sqsubseteq \delta'$ (in $[\![\Delta]\!]$) if for all $\kappa \in \Delta$,

$\delta(\kappa) \leq \delta'(\kappa)$. The intended interpretation of some $\delta \in [\![\Delta]\!]$ is that δ maps each clock κ in Δ to a natural number stating how much time that clock has left to run. The ordering $\delta \sqsubseteq \delta'$ indicates that the clocks in δ have at most as much time left as the clocks in δ'. We use the notation $\delta[\kappa \mapsto n]$ to denote the clock environment mapping κ to n and κ' to $\delta(\kappa')$ when $\kappa \neq \kappa'$.

3.3 Semantic types

We interpret types as $[\![\Delta]\!]$-indexed families of partial equivalence relations (PERs) over the semantic domain D. Recall that a PER on D is a binary relation on D that is symmetric and transitive, but not necessarily reflexive. Since PERs are binary relations, we will treat them as special subsets of the cartesian product $D \times D$. We write $\mathrm{PER}(D)$ for the set of all partial equivalence relations on D, and $\top_D$ for the PER $D \times D$. We require that our $[\![\Delta]\!]$-indexed families of PERs contravariantly respect the partial ordering on elements of $[\![\Delta]\!]$. This ensures that, as the time left to run on clocks increases, the semantic type becomes ever more precise. When we interpret clock quantification as intersection over all approximations, we capture the common core of all the approximations.

Formally, a semantic type for a clock context Δ is a function $A : [\![\Delta]\!] \to \mathrm{PER}(D)$, satisfying contravariant Kripke monotonicity: for all $\delta' \sqsubseteq \delta$, $A\delta \subseteq A\delta'$. We write $\mathrm{ClkPER}(\Delta)$ for the collection of all semantic types for the clock context Δ. In Section 4, we will formally consider morphisms between semantic types, and so turn each $\mathrm{ClkPER}(\Delta)$ into a category. Note that we do *not* require that any of our PERs are admissible. Admissible PERs always include the pair $(\bot, \bot)$, precisely the values we wish to exclude.

We now define semantic counterparts for each of the syntactic type constructors we presented in Section 2.1.

Unit, Product, Coproduct and Function Types The constructions for unit, product and coproduct types are straightforward. The unit type will be interpreted by a constant family of PERs:

$$1\delta = \{(\mathsf{Unit}, \mathsf{Unit})\}$$

This trivially satisfies Kripke monotonicity.

Given semantic types A and B, their product is defined to be

$$(A \times B)\delta = \left\{ \begin{array}{l} (\mathsf{Pair}(x,y), \mathsf{Pair}(x',y')) \\ \quad \mid (x,x') \in [\![A]\!]\delta \wedge (y,y') \in [\![B]\!]\delta \end{array} \right\}$$

and their coproduct is

$$(A + B)\delta = \begin{array}{l} \{(\mathsf{Inl}(x), \mathsf{Inl}(x')) \mid (x,x') \in [\![A]\!]\delta\} \\ \quad \cup \\ \{(\mathsf{Inr}(y), \mathsf{Inr}(y')) \mid (y,y') \in [\![B]\!]\delta\} \end{array}$$

Since A and B are assumed to be semantic types, it immediately follows that $A \times B$ and $A + B$ are semantic types.

To interpret function types, we use the usual definition for Kripke-indexed logical relations, by quantifying over all smaller clock environments. Given semantic types A and B, we define:

$$(A \to B)\delta = \left\{ \begin{array}{l} (\mathsf{Lam}(f), \mathsf{Lam}(f')) \\ \quad \mid \forall \delta' \sqsubseteq \delta, (x,x') \in A\delta'.\ (fx, f'x') \in B\delta' \end{array} \right\}$$

It follows by the standard argument for Kripke logical relations that this family of PERs is contravariant in clock environments.

Guarded Modality Given a semantic type A, we define the semantic guard modality $\triangleright^\kappa$ as follows, where κ is a member of the clock context Δ:

$$(\triangleright^\kappa A)\delta = \left\{ \begin{array}{ll} \top_D & \text{if } \delta(\kappa) = 0 \\ A(\delta[\kappa \mapsto n]) & \text{if } \delta(\kappa) = n+1 \end{array} \right.$$

The semantic type operator $\triangleright^\kappa$ acts differently depending on the time remaining on the clock κ in the current clock environment δ. When the clock has run to zero, $\triangleright^\kappa A$ becomes completely uninformative, equating all pairs of elements in the semantic domain D. If there is time remaining, then $\triangleright^\kappa A$ equates a pair iff A would with one fewer steps remaining. For Kripke monotonicity, we want to prove that $(d,d') \in (\triangleright^\kappa A)\delta$ implies $(d,d') \in (\triangleright^\kappa A)\delta'$ when $\delta' \sqsubseteq \delta$. If $\delta'(\kappa) = 0$ then $(\triangleright^\kappa A)\delta' = D \times D \ni (d,d')$. If $\delta'(\kappa) = n'+1$ then $\delta(\kappa) = n+1$ with $n' \leq n$. So $(d,d') \in A(\delta[\kappa \mapsto n])$, and so by A's Kripke monotonicity, $(d,d') \in A(\delta'[\kappa \mapsto n']) = (\triangleright^\kappa A)\delta'$.

Clock Quantification The semantic counterpart of clock quantification takes us from semantic types in $\mathrm{ClkPER}(\Delta, \kappa)$ to semantic types in $\mathrm{ClkPER}(\Delta)$. Given a semantic type A in $\mathrm{ClkPER}(\Delta, \kappa)$, we define

$$\forall_\kappa A = \bigcap_{n \in \mathbb{N}} A(\delta[\kappa \mapsto n])$$

For Kripke monotonicity, if $(d,d') \in (\forall\kappa.A)\delta$ then $\forall n.\ (d,d') \in A(\delta[\kappa \mapsto n])$. Since A is a semantic type, $\forall n.\ (d,d') \in A(\delta'[\kappa \mapsto n])$, hence $(d,d') \in (\forall\kappa.A)\delta'$.

Complete Lattice Structure In order to define the semantic counterpart of the least fixpoint type operator, we make use of the lattice structure of $\mathrm{ClkPER}(\Delta)$. Given $A, B \in \mathrm{ClkPER}(\Delta)$, we define a partial order: $A \subseteq B$ if $\forall\delta.A\delta \subseteq B\delta$. It is easy to see that $\mathrm{ClkPER}(\Delta)$ is closed under arbitrary intersections, and so is a complete lattice.

Each of the semantic type constructors above is monotonic with respect to the partial order on $\mathrm{ClkPER}(\Delta)$ (with the obvious proviso that $A \to B$ is only monotonic in its second argument).

Least Fixpoint Types We make use of the Knaster-Tarski theorem [24] to produce the least fixpoint of a monotone function on $\mathrm{ClkPER}(\Delta)$. See Loader [17] for an similar usage in a setting without guarded recursion. Given a monotone function $F : \mathrm{ClkPER}(\Delta) \to \mathrm{ClkPER}(\Delta)$, we define:

$$(\mu F) = \bigcap \{A \in \mathrm{ClkPER}(\Delta) \mid FA \subseteq A\}$$

For any monotone F, μF is immediately a semantic type by construction, since semantic types are closed under intersection. As an initial step towards semantic type soundness for our calculus, we demonstrate a semantic well-typedness result for the primrec function we defined in Section 3.1.

Lemma 1. *Let* $F : \mathrm{ClkPER}(\Delta) \to \mathrm{ClkPER}(\Delta)$ *be a monotone function. Let* $\mathrm{fmap} : D \to D \to D$ *be such that for all* $\delta \in [\![\Delta]\!]$, *for all* $A, B \in \mathrm{ClkPER}(\Delta)$, *and for all* $(g,g') \in (A \to B)\delta$ *and* $(x,x') \in FA\delta$, *we have* $(\mathrm{fmap}\ g\ x, \mathrm{fmap}\ g'\ x') \in FB\delta$. *Then, for all* $\delta \in [\![\Delta]\!]$ *and for all* $(f,f') \in (F(\mu F \times C) \to C)\delta$, *where* C *is a semantic type, we have*

$$(\mathrm{primrec}\ \mathrm{fmap}\ f, \mathrm{primrec}\ \mathrm{fmap}\ f') \in (\mu F \to C)\delta$$

Proof. (Sketch) By induction on the least fixpoint $F(\mu F) = \mu F$, using the Knaster-Tarski theorem. □

This lemma only states that our semantic primrec recursion operator is semantically well-typed. We will show in Theorem 2 that primrec also witnesses μF as the carrier of the initial F-algebra, as we claimed in Section 1.4.

3.4 Interpretation of syntactic types

A well-formed type $\Delta; \Theta \vdash A : \mathtt{type}$ is interpreted as monotonic function $[\![A]\!] : \mathrm{ClkPER}(\Delta)^{|\Theta|} \to \mathrm{ClkPER}(\Delta)$, where $|\Theta|$ denotes the number of type variables in Θ. The interpretation is defined in the clauses in Figure 6. These clauses make use of the constructions of semantic types defined in the previous subsection. In the clause for $\forall\kappa$, we make use of the "clock weakening" operator $-\uparrow_\kappa$ which takes a collection of semantic types in $\mathrm{ClkPER}(\Delta)^{|\Theta|}$ to semantic types in $\mathrm{ClkPER}(\Delta, \kappa)^{|\Theta|}$ by pointwise restriction of clock environments in $[\![\Delta, \kappa]\!]$ to clock environments in $[\![\Delta]\!]$.

$$
\begin{array}{rcl}
[\![1]\!]\theta & = & 1 \\
[\![X]\!]\theta & = & \theta(X) \\
[\![A \times B]\!]\theta & = & [\![A]\!]\theta \times [\![B]\!]\theta \\
[\![A + B]\!]\theta & = & [\![A]\!]\theta + [\![B]\!]\theta \\
[\![A \to B]\!]\theta & = & [\![A]\!]\emptyset \to [\![B]\!]\theta \\
[\![\triangleright^\kappa A]\!]\theta & = & \triangleright^\kappa([\![A]\!]\theta) \\
[\![\forall\kappa.A]\!]\theta & = & \forall_\kappa([\![A]\!](\theta{\uparrow_\kappa})) \\
[\![\mu X.F]\!]\theta & = & \mu(\lambda X.[\![F]\!](\theta, X))
\end{array}
$$

Figure 6. Interpretation of well-formed types

The next lemma states that syntactic substitution and semantic substitution commute. The proof is a straightforward induction on the well-formed type A. Note the side condition that $\kappa \notin fc(A)$, matching the side condition on the κ-App typing rule.

Lemma 2. *Assume that* $\Delta; \Theta \vdash A : \mathtt{type}$, $\kappa, \kappa' \in \Delta$ *and* $\kappa' \notin fc(A)$. *Then for all* $\delta \in [\![\Delta[\kappa \mapsto \kappa']]\!]$ *and* $\theta \in \mathrm{ClkPER}(\Delta)^{|\Theta|}$,

$$[\![A]\!]\theta(\delta[\kappa \mapsto \delta(\kappa')]) = [\![A[\kappa \mapsto \kappa']]\!](\theta[\kappa \mapsto \kappa'])\delta.$$

The syntactic type equality judgement $\Delta; \Theta \vdash A \equiv B : \mathtt{type}$ that we defined in Section 2.2 is interpreted as equality of semantic types. The following lemma states that this interpretation is sound.

Lemma 3. *If* $\Delta; \Theta \vdash A \equiv B : \mathtt{type}$ *then for all* $\theta \in \mathrm{ClkPER}(\Delta)^{|\Theta|}$, $[\![A]\!]\theta = [\![B]\!]\theta$.

The next lemma states that we may use the semantic $\mathrm{fmap_F}$ functions defined in Figure 5 with the primrec operator, by showing that $\mathrm{fmap_F}$ always satisfies the hypotheses of Lemma 1.

Lemma 4. *For all* $\Delta; \Theta \vdash F : \mathtt{type}$, *the* fmap_F *defined in Figure 5 are all semantically well-typed: For all* $\delta \in [\![\Delta]\!]$, *for all* $\overrightarrow{A, B} \in \mathrm{ClkPER}(\Delta)$, *and for all* $\overrightarrow{(g, g')} \in \overrightarrow{(A \to B)\delta}$ *and* $(x, x') \in FA\delta$, *we have* $(\mathrm{fmap_F}\ \overrightarrow{g}\ x, \mathrm{fmap_F}\ \overrightarrow{g}\ x') \in FB\delta$.

It may seem that we could prove this lemma by using the syntactic definition of *fmap* from Figure 3 and then applying Theorem 1. However, the semantic well-typedness of our interpretation of primRec depends on the semantic well-typedness of fmap, so we must prove this lemma directly in the model to break the circularity.

3.5 Semantic type soundness

We now state our first main result: the semantic type soundness property of our system. To state this result, we define the semantic interpretation of contexts as a clock-environment indexed collection of PERs over environments:

$$[\![\Gamma]\!]\delta = \{(\eta, \eta') \mid \forall (x : A) \in \Gamma.\ (\eta(x), \eta'(x)) \in [\![A]\!]\delta\}$$

Theorem 1. *If* $\Delta; \Gamma \vdash e : A$, *then for all* $\delta \in [\![\Delta]\!]$ *and* $(\eta, \eta') \in [\![\Gamma]\!]\delta$, $([\![e]\!]\eta, [\![e]\!]\eta') \in [\![A]\!]\delta$.

Proof. (Sketch) By induction on the derivation of $\Delta; \Gamma \vdash e : A$. The most interesting cases are for **fix** and primRec. In essence, the case for **fix** goes through by induction on the value of the counter assigned to the clock variable κ in the current clock environment. The case for primRec is given by Lemma 1 and Lemma 4. □

Corollary 1. *If* $-; - \vdash e : A$ *then for all* η, $[\![e]\!]\eta \neq \bot$.

By this corollary, the denotation of a closed program is never $\bot$, so well-typed programs always yield a proper value. When the result type A is the stream type $\forall\kappa.\mu X.B \times \triangleright^\kappa X$, we can deduce that the denotation of e will always be infinite streams of elements of B. We further elaborate this point in the next section, showing that this type is the carrier of the final $(B \times -)$-coalgebra.

4. Properties of fixpoint types

Using the denotational model of the previous section, we are now in a position to formally state and sketch the proofs of the properties of fixpoint types that we claimed in Section 1.4. Our claimed results are statements in category theoretic language about the initiality and finality of various (co)algebras. Therefore, we first construct suitable categories to work in.

Definition 1. *For each clock variable context* Δ, *the category* $\mathrm{ClkPER}(\Delta)$ *has as objects semantic types over* Δ. *Morphisms* $f : A \to B$ *are continuous functions* $f : D \to D$ *such that for all* $\delta \in [\![\Delta]\!]$ *and* $(a, a') \in A\delta$, $(fa, fa') \in B\delta$. *Two morphisms* f, f' *are considered equivalent if for all* $\delta \in [\![\Delta]\!]$ *and* $(a, a') \in A\delta$, $(fa, f'a') \in B\delta$.

Each well-typed term $\Delta; x : A \vdash e : B$ defines a morphism $[\![e]\!] : [\![A]\!] \to [\![B]\!]$ in $\mathrm{ClkPER}(\Delta)$. We can define equality between terms in our syntactic type system in terms of the equality on morphisms in this definition. Moreover, each well-formed type operator $\Delta; X \vdash F[X] : \mathtt{type}$ defines a (strong) realisable endofunctor on $\mathrm{ClkPER}(\Delta)$ that is monotonic on objects, using the semantic fmap_F defined in Figure 5 to define the action on morphisms. We have already checked (Lemma 4) that this is well-defined, and it is straightforward to check that the usual functor laws hold. In what follows, whenever we refer to an endofunctor on $\mathrm{ClkPER}(\Delta)$, we mean a realisable functor that is monotonic on objects, and we will use $fmap_F$ to refer to the action of a functor F on morphisms.

Initial Algebras Recall that, for any functor F, an F-algebra is a pair of an object A and a morphism $k : FA \to A$. A homomorphism h between (A, k_A) and (B, k_B) is a morphism $h : A \to B$ such that $h \circ k_A = k_B \circ fmap_F\ h$. An *initial* F-algebra is an F-algebra (A, k_A) such that for any other F-algebra (B, k_B), there is a unique homomorphism $h : (A, k_A) \to (B, k_B)$.

Theorem 2. *If* F *is an endofunctor on* $\mathrm{ClkPER}(\Delta)$, *then* μF *is the carrier of an initial* F*-algebra.*

Proof. (Sketch) Since μF is a fixpoint of F, the morphism $F(\mu F) \to \mu F$ is simply realised by the identity map. Given any other F-algebra (B, k_B), define a morphism $\mu F \to B$ using the primrec operator from Lemma 1. Checking that this gives an F-algebra homomorphism is straightforward, proving uniqueness uses induction on elements of μF, by the Knaster-Tarski theorem. □

Guarded Final Co-Algebras Theorem 2 applies to all functors F, and in particular functors of the form $F(\triangleright^\kappa -)$ on the category $\mathrm{ClkPER}(\Delta, \kappa)$. As well as $\mu(F(\triangleright^\kappa -))$ being the carrier of an initial algebra, it is also the carrier of a final $F(\triangleright^\kappa -)$-coalgebra.

Coalgebras are the formal dual of algebras: for an endofunctor F, an F-coalgebra is a pair of an object A and a morphism $k_A : A \to FA$. A homomorphism $h : (A, k_A) \to (B, k_B)$ of coalgebras is a morphism $h : A \to B$ such that $fmap_F\ h \circ k_A = k_B \circ h$. A *final* F-coalgebra is an F-coalgebra (B, k_B) such that for any other F-coalgebra (A, k_A), there is a unique F-coalgebra homomorphism $\mathrm{unfold}\ k_A : (A, k_A) \to (B, k_B)$.

Theorem 3. *If* F *is an endofunctor on* $\mathrm{ClkPER}(\Delta)$, *then* $\mu(F(\triangleright^\kappa -))$ *is the carrier of a final* $F(\triangleright^\kappa -)$*-coalgebra in* $\mathrm{ClkPER}(\Delta, \kappa)$.

Proof. (Sketch) As for Theorem 2, since $\mu(F(\triangleright^\kappa -))$ is a fixpoint of $F(\triangleright^\kappa -)$, the morphism $\mu(F(\triangleright^\kappa -)) \to F(\triangleright^\kappa(\mu(F(\triangleright^\kappa -))))$ is simply realised by the identity map. Given any other F-coalgebra (A, k_A), define a morphism $\mathrm{unfold}\ k_A : A \to \mu(F(\triangleright^\kappa -))$ as $\mathrm{fix}\ (\lambda g\ a.\ \mathrm{fmap}_F\ g\ (k_A\ a))$. It is straightforward to prove that this

is an F-coalgebra homomorphism. Uniqueness is proved for each possible clock environment δ by induction on $\delta(\kappa)$. □

The syntactic counterpart of the construction we used in this proof is exactly the term we used in Section 1.4 for the definition of *unfold*. It is also easy to check that the term *deCons* we defined there is semantically equivalent to the identity. Therefore, Theorem 3 substantiates the claim we made in Section 1.4 that $\mu X.F[\triangleright^\kappa -]$ is the syntactic description of the carrier of a final $F[\triangleright^\kappa -]$-coalgebra.

Final Co-Algebras Theorem 3 gives final coalgebras in the categories $\mathrm{ClkPER}(\Delta, \kappa)$, where we have a spare clock variable. By using clock quantification, we can close over this clock variable, and get the final F-coalgebra, not just the final $F(\triangleright^\kappa -)$-coalgebra.

Theorem 4. *For an endofunctor F on $\mathrm{ClkPER}(\Delta)$, $\forall_\kappa.\mu(F(\triangleright^\kappa -))$ is the carrier of a final F-coalgebra in $\mathrm{ClkPER}(\Delta)$.*

Proof. (Sketch) Almost identical to the proof for Theorem 3. □

This final theorem, along with the examples we presented in Section 2.5, substantiates our claim that the combination of clocks and guards that we have presented in this paper is a viable and comfortable setting in which to productively coprogram.

5. Conclusions and Further Work

We have presented a semantics for a small total calculus with primitive recursion for inductive data and a compositional treatment of corecursion, ensuring causality via the applicative structure of a *local* notion of time. In effect, we use time-based typing to grow a given total language, where all computation terminates within one 'day', into a larger total language, where additional recursion is justified clearly by the advancing clock. Functions from clocked inputs to clocked outputs enforce precise producer-consumer contracts—today's output must be computed only from today's input—documenting their utility as components of productive processes. Quantifying clock variables localises the time stream to a particular construction whose clients can then use it 'in the moment'. The method, made local, can be iterated, with inner clocks justifying the totality of computations within one 'day' of an outer clock.

At present, however, we have used local time only to justify productive corecursion, with only primitive recursion for inductive types. It seems pressing to ask whether local time might similarly liberalise termination checking, with a local clock measuring time into the past and ensuring that recursive calls receive old enough inputs that their outputs are ready when we need them. We are actively seeking a semantics for such a system, but it currently seems more difficult to pin down.

In due course, we should like to grow this experimental calculus to a full blown dependent type theory where (co)recursive constructions are checked to be total within nested local time streams, then exported to their clients without clocking. At least we have now established what local time streams are and how to extract productive processes from them.

Acknowledgements We would like to thank Lars Birkedal, Rasmus Møgelberg, and Paula Severi for extremely useful comments and discussions.

References

[1] A. Abel. Termination checking with types. *ITA*, 38(4):277–319, 2004.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[3] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21:239–250, 1984.

[4] L. Birkedal and R. E. Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *ACM/IEEE Symposium on Logic in Computer Science (LICS 2013)*, 2013.

[5] L. Birkedal, J. Schwinghammer, and K. Støvring. A metric model of guarded recursion. In *Presented at FICS 2010*, 2010.

[6] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Meth. in Computer Science*, 8(4), 2012.

[7] V. Capretta. General recursion via coinductive types. *Log. Meth. in Computer Science*, 1(2), 2005.

[8] N. A. Danielsson. Beating the productivity checker using embedded languages. In *Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010)*, volume 43 of *EPTCS*, pages 29–48, 2010.

[9] N. A. Danielsson and T. Altenkirch. Mixing induction and coinduction. Draft, 2009.

[10] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Log. Meth. in Computer Science*, 7(2), 2011.

[11] N. Ghani, P. Hancock, and D. Pattinson. Representations of stream processors using nested fixed points. *Log. Meth. in Computer Science*, 5(3), 2009.

[12] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs, International Workshop TYPES'94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59, 1994.

[13] G. Hutton and M. Jaskelioff. Representing contractive functions on streams. Submitted, 2011.

[14] N. R. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, pages 45–57, 2011.

[15] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *IEEE Symposium on Logic in Computer Science, LICS 2011*, pages 257–266, 2011.

[16] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 24–35, 1994.

[17] R. Loader. Equational Theories for Inductive Types. *Annals of Pure and Applied Logic*, 84(2):175–217, 1997.

[18] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Prog.*, 18(1):1–13, 2008.

[19] E. Moggi and A. Sabry. Monadic encapsulation of effects: a revised approach (extended version). *J. Funct. Prog.*, 11(6):591–627, 2001.

[20] H. Nakano. A modality for recursion. In *IEEE Symposium on Logic in Computer Science (LICS 2000)*, pages 255–266, 2000.

[21] A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Mathematical Foundations of Programming Semantics, Proc. 9th Int. Conf.*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, Berlin, 1994.

[22] P. Severi and F.-J. de Vries. Pure type systems with corecursion on streams: from finite to infinitary normalisation. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, 2012.

[23] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

[24] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.