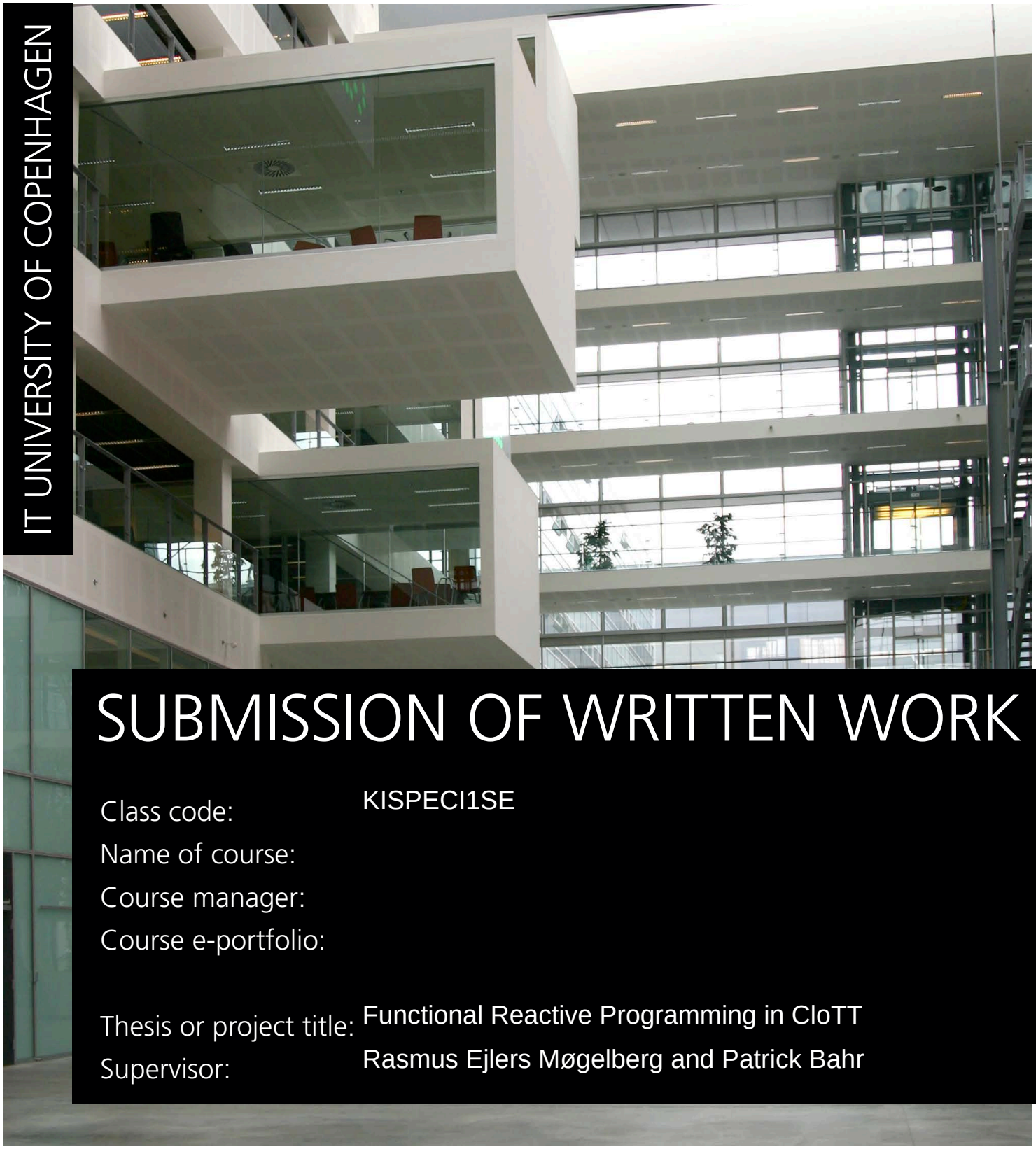
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code: KISPECI1SE

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title: Functional Reactive Programming in CloTT

Supervisor: Rasmus Ejlers Møgelberg and Patrick Bahr

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Adam Bjørn Schønemann	03/04/1991	adsc@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Abstract

Guarded recursion allows a form of general recursion to be added to a language without breaking consistency. Consistency is especially important for proof-assistants, but is also a useful property for safe general-purpose programming. Guarded recursion has proven a natural fit for the domain of FRP (functional reactive programming), since FRP programs are inherently required to be both causal and productive; both properties are elegantly ensured by guarded recursion. Recently, several type theories based on guarded recursion have emerged, one of which is CloTT. CloTT uses guarded recursion to also type definitions that are *acausal* yet productive, by using *clock-variables* and *clock-quantification*. Thus, it extends the set of definitions that can be typed while still ensuring consistency in the context of proof-assistants. Such acausal definitions could also prove useful in the domain of FRP, as long as a clear distinction between causal and acausal functions is maintained, which CloTT luckily does. This thesis presents an implementation of the simply-typed fragment of CloTT which is aimed at practical programming with both inductive, guarded-recursive and coinductive types. We expand the calculus with useful constructs; design and implement a modern type-inference system with user-definable types; design a big-step operational semantics and implement an interpreter from it. We evaluate the interpreter empirically in order to explore the runtime characteristics of well-typed programs in our language. We conclude that there is nothing inherent to clock-quantification that complicates implementations significantly, but that practical limitations of the semantics rule out the desired runtime behaviour for certain classes of programs. Finally, we present some ideas that we theorize could solve these limitations in an elegant and simple way.

Acknowledgements

I would like to thank my supervisors Rasmus Møgelberg and Patrick Bahr for giving me this opportunity to write my thesis on a subject that is interesting, and allows me to explore programming language design and implementation in Haskell, and for guiding me through the process and helping me understand the subject.

I would also like to thank my supporting family, especially my girlfriend who have had to keep the house together while I've sat late hours isolated at the computer frowning for hours on end.

Preface

This thesis describes the design and implementation of a programming language; the code can be found in a git repository at https://github.com/adamschoenemann/clofrp. This report is rather long, measuring over 80 pages. However, despair not! Much of the content is figures, rules and code examples, so the actual textual content does not constitute more than roughly half of those pages. Although this is a one-person effort, I shall use the editorial "we" as the personal pronoun throughout the report, as it is a bit more neutral, and feels more appropriate.

Functional Reactive Programming in CloTT

Master's Thesis

Adam Schønemann, adsc@itu.dk

Contents

List of Figures

List of Listings

1 Introduction

Guarded recursion was originally developed in the context of proof-assistants, but lately its applicability to other domains, especially FRP, has been discovered. We have in previous work [1] implemented a language based on that of Krishnaswami [2] who used guarded recursion to design an operational semantics for a FRP language that statically prevents space-leaks, a problem that is often critical to exactly the kind of reactive programs that FRP is designed to implement. While this implementation served well as a proof of concept, it lacked the features necessary for practical programs to be constructed in it. Many features could be added without significant problems, but one particular challenge is harder to address: Krishnaswami's language exclusively permits guarded-recursive types, which is overly restrictive in a practical setting. We would like to work with both inductively and coinductively defined data as well, in a way that safely allows us to mix and match between the different flavours of recursive types. The recent work on clock-quantification, a concept originally due to Atkey and McBride [3], proposes a way to accomplish this. CloTT [4] expands on this work, and gives a small-step operational semantics for a dependently-typed language with clock-quantification. Therefore, we use a simply-typed version of CloTT as our starting point in order to explore how clock-quantification and guarded recursion integrates in a practical setting, with added emphasis on FRP.

1.1 Problem statement

The goal of this project is to produce a practical implementation of the simply-typed version of the CloTT [4] language, primarily targeted at the domain of functional reactive programming, in order to explore how the theory of CloTT integrates with a practical setting.

Thus, the scope of the project can be outlined as:

1. Implement a parser and a type-inference and type-checking algorithm that facilitates ergonomic authoring of simply-typed CloTT programs.
2. Formulate a big-step operational semantics for the language.
3. Implement an interpreter for CloTT programs in Haskell based on the operational semantics.
4. Design and implement an API for inter-operation of CloTT and Haskell programs
5. Demonstrate the implementation by implementing some simple programs and showing how they behave at runtime.
6. Compare with related work, specifically that of [2].

Note that this project started out with a much stronger emphasis on the comparison with the work of [2]. While a such a comparison is still warranted and included, it became clear during the course of the project that it is not the most interesting contribution.

2 Motivation and Related Research

Developing interactive programs pose a lot of new challenges compared to traditional batch programs. There is a continuous feedback-loop of user-input and program-output instead of well-defined start- and end-points and thus the programs do not terminate by design, except when the user signals termination. Traditional programming languages do not map naturally to this domain, as they are built around a sequential approach to communication, ie. function calls and control flow primitives.

As a consequence, to construct interactive programs, it is necessary for the different parts of the program to communicate with each-other using callbacks, shared mutable state and concurrency. These are all things that are very complicated to get right, especially as the interactive programs we wish to develop grow in scale and complexity. Furthermore, attempting to verify such programs using formal methods is incredibly hard.

2.1 Functional Reactive Programming

One proposal to fix this is a ”new” paradigm of programming called *Functional Reactive Programming*, first proposed in 1997 [5]. It has gained some popularity in recent years [6–8], but is still far from mainstream.

FRP models inputs and outputs as streams, or signals, of data. Fundamentally, a stream represents a value that varies over time. Signals can then be processed and manipulated using traditional constructs from functional programming. Interactive programs then become ”internally pure” and interact with the outside world at clearly defined boundaries. However, traditional FRP programming comes with several pitfalls:

1. *Causality* – Traditional “naïve” FRP does not enforce causality, i.e. that output at time n only depends on input at and before time n [2]. Since time is modeled as a stream, nothing is stopping you from looking ”ahead” into the stream, which is not semantically well-founded. For example `noncausal (x :: y :: xs) = y :: noncausal xs`, where ’::’ is list-cons, will always depend on a future value before producing any output, and thus cannot produce output in the current time-step.
2. *Productivity* – Since FRP programs are inherently non-terminating, structural recursion is not useful. However, recursive definitions must still produce output in finite time (productivity), even if they are non-terminating. A program such as `zeros = 0 :: zeros` is well-defined, yet another program `xs = xs` is not. How do we tell the difference?
3. *Resource usage* – Since FRP programs abstract away resource usage, yet captures time-dependencies between values implicitly, it is easy to inadvertently define a function that depends on the entire past, thus accumulating the entire history of a stream, resulting in a memory leak. For example, `const xs = xs :: const xs` will accumulate the entire history of the `xs` stream, and will at some point run out of memory. However, if `xs` was a non-time dependent value, like a number, this would be totally fine.

Several approaches to solving these problems have been suggested, among them *event-driven FRP* [9] and *arrowized FRP* [10]. Common to all of these approaches is that they limit the expressiveness of classical FRP in order to prevent some or all of these pitfalls. This is typically done by demoting streams from being first-class values, and then exposing some pre-defined combinators in order to construct functions that work on streams. Typically, this entails prohibiting changing the structure of the signal graph at runtime, and thus also all higher-order functions (e.g. of type stream-of-streams). However, these limitations are often to such a degree that they significantly reduce the expressiveness of the FRP paradigm. To regain dynamic behaviour, like switching between two streams, they expose primitive combinators. However, these combinators re-introduce the possiblity of memory issues and non-productivity, and furthermore have an ad-hoc flavour to them [2].

There are also approaches that attempt to exploit the type-system to statically ensure causality and productivity, using a technique called *Guarded Recursion*.

2.2 Guarded Recursion

Guarded recursion was originally suggested by Nakano [11] to address the challenge of working with coinductive data in proof assistants such as Coq [8] and Agda [12]. Guarded recursion introduces a new modality $\triangleright$ pronounced “later” which makes it possible to distinguish between values we can use now, and values we can only use later. Coincidentally, this distinction is exactly what is needed in order to ensure causality, by constructing a type-system that prohibits using “later” values now. In the same way, guarded recursion ensures productivity, by requiring that definitions return something “now” if they are to be used “now”. Several approaches for using guarded recursion for FRP programming have been proposed [2, 13, 14]. For all intents and purposes, guarded recursion seems to be a natural fit for FRP.

The third pitfall is still unadressed: resource usage. Krishnaswami et al. proposed using linear types in order to fix this [15], but found that the resulting types became too complicated and inflexible to be usable in practice, as the exact size of the data-flow graph is reflected in the type of a program. In [2],

Krishnaswami presented a new programming language that – by using guarded recursion, a specific operational semantics, and a modified, more relaxed approach to restricting memory allocation – set out to solve all three of the above mentioned pitfalls. His operational semantics is quite simple to implement (which we did, in a previous project [1]), but the flipside is that the language is not particularly simple to program in, as one has to be very explicit about when to retain values across time steps. Since we will later compare this project with our previous implementation of Krishnaswami's work (a language we chose to call ModalFRP), we shall give a short exposition on his theory from [2], which is adapted from our previous report [1].

2.3 ModalFRP

The principle of ModalFRP lies in its operational semantics: Programs are evaluated over time in so called "ticks". During one tick, everything is evaluated using call-by-value, so no thunks are generated or built up. However, the programmer can *explicitly* choose to delay a computation to the next time step. Such a delayed computation is allocated in a store. After a tick has completed, all values that are allocated on the store which were available at the tick are simply deleted, and can thus never be used again. All expressions that were delayed on the store are advanced, and are therefore available for the next tick. The evaluation of the program advances by shifting between executing the program in a store σ, ticking the store to σ' and then evaluating the program in the new store. This combination of call-by-value (eager) and call-by-need (lazy) evaluation ensures that neither time-leaks or space-leaks are possible. However, a new typing discipline is required to ensure that programs do not reference deleted values on the store.

The types and terms of ModalFRP closely resemble the simply-typed lambda calculus, but extends it with the modal types $\Box$ ("stable") and $\bullet$ ("later" or "next"). These modalities denote subsets of types whose values are always available, or not availble until the text time-step, respectively. Values of a type without any modality are only available now. Furthermore, the temporal recursive type $\widehat{\mu}\alpha.\ A$ is introduced, which defines types that are recursive through time, and a fixpoint for guarded recursion in the style of Nakano [11] is also provided to populate the $\widehat{\mu}$ types.

We shall not re-produce the full grammar, semantics and typing rules here. Instead we shall focus on a few of the interesting parts that separate the language from the standard lambda calculus (Figure 1).

$$
\begin{array}{llll}
\text{Types} & A & ::= & \ldots \mid \bullet\, A \mid \Box\, A \mid \widehat{\mu}\alpha.\ A \mid S\ A \mid \texttt{alloc} \\
\text{Terms} & e & ::= & \ldots \mid \delta_{e'}(e) \mid \texttt{let}\ \delta(x) = e\ \texttt{in}\ e' \mid \texttt{stable}(e) \mid \texttt{let stable}(x) = e\ \texttt{in}\ e' \mid \\
 & & & \texttt{into}\ e \mid \texttt{out}\ e \mid \texttt{cons}(e, e') \mid \texttt{let cons}(x, xs) = e\ \texttt{in}\ e' \mid \texttt{promote}(e) \\
 & & & \texttt{l} \mid \texttt{!l} \mid \diamond \\
\text{Values} & v & ::= & \ldots \mid \texttt{l} \mid \diamond \mid \texttt{stable}(v) \mid \texttt{cons}(v, v') \\
\text{Stores} & \sigma & ::= & \cdot \mid \sigma, \texttt{l} : v\ \texttt{now} \mid \sigma, \texttt{l} : e\ \texttt{later} \mid \sigma, \texttt{l} : \texttt{null}
\end{array}
$$

Figure 1: Syntax extensions to the simply-typed lambda calculus.

As mentioned above, the standard types are augmented with the $\bullet$ and $\Box$ modalities, as well as the temporal recursive type. Added to this is a primitive type of streams S and the type of allocation tokens `alloc`. Functions that have an allocation token in scope can delay values into the next time-step by allocating them on the store.

Terms include an introduction and elimination form of the $\bullet$ and $\Box$ modalities (δ and `stable` respectively), as well as temporal recursive types (`into` and `out`). Primitives for constructing and deconstructing streams are also added. `promote` takes a term, and if it is of an inherently stable type, like $\mathbb{N}$, we can "promote" it to be `stable`. There is syntax for pointers, represented as `l` for a pointer label and `!l` for a pointer dereference. However, pointers are not surface syntax, and as such can never appear in a user-program. Instead, pointers are used by the evaluator to suspend computations to a later time. Finally, the $\diamond$

represents an allocation token, which is also not part of the surface syntax – only the runtime can create allocation tokens.

The typing context Γ of ModalFRP does not simply map names to types, but names to a type paired with a temporal qualifier (Figure 2).

$$\begin{array}{lllll} \text{Qualifiers} & q & ::= & \texttt{now} \mid \texttt{stable} \mid \texttt{later} \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A\ q \end{array}$$

Figure 2: Qualifiers and typing contexts.

These qualifiers encode when a value can be typed. For example, to delay a value into the next time step, we must type-check it in a context that only retains stable values and values that are also delayed. This can be seen when type-checking $\delta_{e'}(e)$ and $\texttt{let}\ \delta(x) = e\ \texttt{in}\ e'$ (Figure 3). There are similar rules for the `stable` forms, which further restrict the context to only contain values qualified with `stable`. These typing rules serve to only type programs that respect causality, as in "expressions executed in the future can only depend on expressions in the future" and "expressions that are time-independent can only rely on other time-independent expressions".

$$\frac{\Gamma \vdash e : A\ \texttt{later} \qquad \Gamma \vdash e' : \texttt{alloc now}}{\Gamma \vdash \delta_{e'}(e) : \bullet A\ \texttt{now}} \bullet \mathrm{I} \qquad \frac{\Gamma \vdash e : A\ \texttt{now} \qquad \Gamma, x : A\ \texttt{later} \vdash e' : C\ \texttt{now}}{\texttt{let}\ \delta(x) = e\ \texttt{in}\ e' : C\ \texttt{now}} \bullet \mathrm{E}$$

$$\frac{\Gamma^\bullet \vdash e : A\ \texttt{now}}{\Gamma \vdash e : A\ \texttt{later}} \mathrm{TL}_{\mathrm{ATER}}$$

$$\begin{array}{lcl} (\cdot)^\bullet & = & \cdot \\ (\Gamma, x : A\ \texttt{later})^\bullet & = & \Gamma^\bullet, x : A\ \texttt{now} \\ (\Gamma, x : A\ \texttt{stable})^\bullet & = & \Gamma^\bullet, x : A\ \texttt{stable} \\ (\Gamma, x : A\ \texttt{now})^\bullet & = & \Gamma^\bullet \end{array}$$

Figure 3: Rules and context-operations for delaying values.

Notice that $\bullet$I requires an allocation token to delay a term. This makes sure that all definitions that delay values, and thus extend the store, are marked by having `alloc` as a parameter in their type. As such, `alloc` represents a form of capability. Furthermore, the allocation tokens serve to prevent construction of values of type $\Box \bullet A$ (e.g. $\texttt{stable}(\delta_u(42))$) – since values of $\bullet A$ are stored on the heap, they are deleted in 2 time-steps, and thus should not be allowed to be marked as stable.

$$\frac{\Gamma \vdash e : [\bullet(\hat{\mu}\alpha.\ A)/\alpha]A\ \texttt{now}}{\Gamma \vdash \texttt{into}\ e : \hat{\mu}\alpha.\ A\ \texttt{now}} \mu\mathrm{I} \qquad \frac{\Gamma \vdash \hat{\mu}\alpha.\ A\ \texttt{now}}{\Gamma \vdash \texttt{out}\ e : [\bullet(\hat{\mu}\alpha.\ A)/\alpha]A\ \texttt{now}} \mu\mathrm{E} \qquad \frac{\Gamma^\Box, x : A\ \texttt{later} \vdash e : A\ \texttt{now}}{\Gamma \vdash \texttt{fix}\ x.\ e : A\ \texttt{now}} \mathrm{Fix}$$

Figure 4: Typing rules for temporal recursive types and fixpoints.

As figure 4 illustrates, the (un)folding of a temporal recursive type mirrors a normal recursive type, except that the substituted type-variable is always guarded in a delay modality. This ensures that only one "step" of the recursive value is unfolded at each tick. Similarly, the judgment for the fixpoint ensures that the recursive call is always guarded, since a value of type $A : \texttt{later}$ can only be used in a delayed expression. As a fixpoint will be evaluated at multiple time-steps, it cannot capture any time-dependent expressions in its closure – the $\Gamma^\Box$ context ensures this.

The semantics of ModalFRP are for the most part fairly standard call-by-value semantics. However, expressions can be delayed to the next tick, by using the introduction form of the $\bullet$ modality ($\delta_{e'}(e)$). Semantically, this is modelled by allocating a new expression on the store and returning a pointer to the expression in the store. Figure 5 show the semantics involving delayed expressions.

$$\frac{\langle \sigma, e' \rangle \Downarrow \langle \sigma', \diamond \rangle \qquad \mathtt{l} \notin \mathrm{dom}(\sigma')}{\langle \sigma, \delta_{e'}(e) \rangle \Downarrow \langle \sigma', \mathtt{l} : e \ \mathtt{later} \ ; \mathtt{l} \rangle} \qquad \frac{\langle \sigma; e \rangle \Downarrow \langle \sigma'; \mathtt{l} \rangle \qquad \langle \sigma; [!\mathtt{l}/x]e' \rangle \Downarrow \langle \sigma''; v \rangle}{\langle \sigma; \mathtt{let} \ \delta(x) = e \ \mathtt{in} \ e' \rangle \Downarrow \langle \sigma''; v \rangle} \qquad \frac{\mathtt{l} : v \ \mathtt{now} \in \sigma}{\langle \sigma; !\mathtt{l} \rangle \Downarrow \langle \sigma; v \rangle}$$

Figure 5: Semantics for delayed expressions.

The two examples in Figures 6 and 7 show two programs in the calculus of [2]: one that type-checks and one that does not.

```
const : S alloc → ℕ → S ℕ
const us n =
  let cons(u,δ(us')) = us in
  let stable(x) = promote(n) in
  cons(x, δᵤ (const us' x))
```

Figure 6: The constant function that repeats a natural number.

```
scary_const : S alloc → S ℕ → S (S ℕ)
scary_const us ns =
  let cons(u,δ(us')) = us in
  let stable(xs) = promote(ns) in -- TYPE ERROR
  cons(xs, δᵤ (scary_const us' xs))
```

Figure 7: The const function that cannot be typed since it causes a memory leak. Note that the only difference lies in the type signature – we cannot constantly repeat time-varying values without causing memory leaks.

2.4 Clocked Type Theory

CloTT (Clocked Type Theory) is another programming language with guarded recursion, that has been developed by Bahr, Grathwohl and Møgelberg [4]. It is a result of a line of research [3, 16, 17] revolving around ensuring productivity by modeling coinductive types using guarded recursive types. While CloTT has not been formulated for FRP specifically, it lends itself well to this problem domain. In comparison to Krishnaswami's language, CloTT introduces the concept of clock variables, which abstracts over time and allows non-causal definitions that cannot be expressed in Krishnaswami's language. Of course, disallowing non-causal definitions is a property that we are very interested in when it comes to FRP, yet this causality requirement is only necessary when dealing with temporal recursive types, i.e. types whose values are a function of time. Thus, guarded recursion alone prevents us from modeling "truly" coinductive data. We shall later explain CloTT in more detail to see how it addresses this challenge.

There is a reduction semantics for CloTT [4], but not an operational big-step semantics. CloTT implicitly retains values through time-steps, and thus do not provide any guarantees with regards to space/time leaks.

CloTT uses higher-rank types to model coinductive data-types. Traditional inference systems using variations of the Hindley-Milner inference [18] are not capable of dealing adequately with such higher-rank types. Therefore, we shall seek to implement a form of type-inference that can deal with these types in a natural way. We shall deal with CloTT quite a bit in this report, so Section 3 is dedicated to describing it.

2.5 Type-inference

While it is perfectly possible to program with a fully annotated calculus such as CloTT, it is quite a burden on the programmer to explicitly annotate every term with its type. It quickly becomes infeasible

to encode anything non-trivial, and the semantics of programs becomes undecipherable because it is buried under type signatures. Therefore, much effort has been put into inferring the types of programs in order to (partially) eliminate this burden from the programmer.

The most famous inference system is the Damas-Hindley-Milner system [18, 19]. This inference system has the property that it is both easy to understand and implement, and infers principal types (that is, the most general type of an expression). The result is an inference system where the programmer can entirely omit writing any types for any expression. Implementations of DHM inference have seen wide industrial use in, among others, Standard ML, OCaml and Haskell. However, for all the power and simplicity that DHM brings, it is fragile and breaks down in the presence of many modern type-system features, such as higher-rank types, generalized algebraic types or impredicative polymorphism. Especially higher-rank types are important for this project, since coinductive types are modelled with such types. While many attempts to extend traditional DHM systems with such features have been made, many suffer from awkward edge-cases or complicated specifications and/or implementations [20–24]. An implementation of the system described in [24] actually made it into the Haskell compiler where it was used for years (it has now been replaced with the "OutsideIn(X)" algorithm [25]). Noticeably, [24] combines DHM inference with a bidirectional [26] system, that requires the programmer to annotate some functions, but in return propagates these annotations down the syntax tree, which results in large degrees of inference for a few annotations.

Bidirectional type-inference has risen in popularity in recent years, and has been used with success to provide inference for even quite complicated type-systems, including dependent types [27–29], subtyping [26], refinement types [30], termination checking [31] and even object-oriented languages [32], proving that it scales well to advanced type-systems. It has also been described as being relatively easy to implement [30] and specify, and more readily gives better error messages [24], an issue that DHM systems traditionally have been struggling with [33].

More recently, Dunfield and Krishnaswami presented a bidirectional inference algorithm that is relatively easy to implement and understand, and has first-class support for higher-rank types [30]. It is this algorithm that we will take as our basis for this project. Note that they later extended their algorithm to handle generalized algebraic data-types in [30], which on the other hand considerably complicated the presentation. We shall later present the main points of the "easy" algorithm in Section 5.1.

3 CloTT

This section will briefly present the theory behind CloTT. For a more complete presentation, refer to [4]. The following paragraph from [4] introduces CloTT:

> Clocked Type theory is an extension of dependent type theory with a special collection of sorts called *clocks*. An inhabitant of a clock is referred to as a *tick*, and these are resources that can be used to unfold fixed point definitions or fold or unfold elements of recursive types. We use κ to range over clock variables and α to range over ticks, so that an assumption of the form $\alpha : \kappa$ states that α is assumed to be a tick on clock κ. Clock contexts Δ are finite sets of clock variables and in our syntax these are given a special role by being subscript to the turnstile as in e.g. typing judgments $\Gamma \vdash_\Delta t : A$. The calculus can be alternatively presented without the subscript by using assumptions of the form $\kappa : \mathsf{Clock}$ in Γ, but we choose the current presentation because it is closer to the semantics given in the strong normalisation proof below.

Clocks are not types, just names, and thus have no constructors, and declarations like $\alpha : \kappa \to \kappa'$ are not allowed. Ticks on a clock that appear in a context such as $\Gamma, \alpha : \kappa, \Gamma' \vdash_\Delta$ represents that a tick on the clock κ occurs between when Γ and Γ'. Thus, values to the left of the tick must be available for one more time step.

The type $\triangleright(\alpha : \kappa).A$ is the type of suspended computations that yield an element of A when it is given a tick α on the clock κ. If α is not free in A (it does not appear unbound), then we simply write $\triangleright^\kappa A$ for $\triangleright(\alpha : \kappa).A$.

A suspended computation represented by a closed term of type $\triangleright(\alpha : \kappa).A$ can be forced by applying it to the tick constant $\diamond$. The rule for this operation can be seen in Figure 8.

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright(\alpha : \kappa).A \qquad \Gamma \vdash_{\Delta} \qquad \kappa' \in \Delta}{\Gamma \vdash_{\Delta} (t[\kappa'/\kappa])[\diamond] : A[\kappa'/\kappa][\diamond/\alpha]}$$

Figure 8: Tick-application rule in CloTT.

Figure 8 says that we can force a suspended computation $t : \triangleright(\alpha : \kappa).A$ to $t[\kappa'/\kappa] : A[\kappa'/\kappa]$ by applying the tick constant $\diamond$. Importantly, this can only be done when Γ is a context where κ does not appear free, that is, it is a well-formed context without mentioning κ, and thus κ-stable, i.e. not affected by ticks on clock κ. κ' is substituted for κ in t and A in order to make the rule closed under substitution and weakening.

The type $\forall\kappa.A$ is a dependent product over clocks. It basically says that a value of such a type can compute an A for any arbitrary number of ticks on the clock κ. We shall later see that such a type (which guarantees productivity) can be used to "escape" the guarded world in a consistent manner, and convert the type to a coinductive type.

We can "force" a value of $\forall\kappa.\ \triangleright(\alpha : \kappa).A$ to $\forall\kappa.A$, if α is not free in A. This basically says that "if we have a computation that for any κ will yield an A in one tick extra, then we can instantiate it with any κ that has at least one more time step left to run and then extract it".

There is a *delayed fixed point* dfix^{κ} which, given a productive function $f : \triangleright^{\kappa}A \to A$ applies f to itself when given a tick. Figure 9 formalizes this.

$$\frac{\Gamma \vdash_{\Delta} f : \triangleright^{\kappa}A \to A}{\Gamma \vdash_{\Delta} \mathsf{dfix}^{\kappa} f : \triangleright^{\kappa}A}$$

Figure 9: The rule for delayed fixpoints in CloTT.

Since CloTT is dependently typed, recursive types are also formulated using dfix. However, since this thesis will be implementing the simply-typed version of CloTT, we shall have to formulate a new rule to encode guarded recursive types.

3.1 Reduction semantics

To reduce a program in CloTT, one must provide a tick as "fuel" to unfold fixpoints. Tick variables can be understood as placeholders for the actual tick $\diamond$ – they represent a promise of a tick, and thus cannot be used for unfolding until they are instantiated with a "concrete" tick. Figure 10 shows an excerpt of the reduction semantics.

$$\begin{array}{rcl@{\qquad}rcl}
(\lambda x : A.t)s & \to & t[s/x] & (\Lambda\kappa.t)[\kappa] & \to & t[\kappa'/\kappa] \\
(\lambda(\alpha' : \kappa').t)[\alpha] & \to & t[\alpha/\alpha'] & \lambda(\alpha : \kappa).(t[\alpha]) & \to & t \\
(\Lambda\kappa.t[\kappa]) & \to & t & \pi_i\langle t_1, t_2\rangle & \to & t_i \\
\mathsf{fold}_{\diamond} t & \to & t & \mathsf{unfold}_{\diamond} t & \to & t \\
\mathsf{if\ true}\ t_1 t_2 & \to & t_1 & \mathsf{if\ false}\ t_1 t_2 & \to & t_2 \\
\mathsf{rec}\ (\mathsf{suc}\ t_1)\ t_2\ t_3 & \to & t_3\ t_1\ (\mathsf{rec}\ t_1\ t_2\ t_3) & \mathsf{rec}\ 0\ t\ s & \to & t \\
(\mathsf{dfix}^{\kappa}\ t)[\diamond] & \to & t\ (\mathsf{dfix}^{\kappa}\ t) & & &
\end{array}$$

Figure 10: Reduction semantics for CloTT.

The η reductions are subject to the usual side-condition that α is not in the set of free tick-variables of t, and κ is not in the set of free clocks of t.

3.2 An example

The following example from [4] shows how to encode coinductive streams in CloTT. To do so, we shall use the fixpoint dfix^κ on the type level to model guarded recursive types, which is possible since CloTT has dependent types. Let $F : \triangleright^\kappa \boldsymbol{\mathcal{U}} \to \boldsymbol{\mathcal{U}}$ be defined as $\lambda(x : \triangleright^\kappa \boldsymbol{\mathcal{U}}).\ \mathsf{Nat} \mathbin{\hat{\times}} (\hat{\triangleright}\,(\alpha : \kappa).\ x[\alpha])$. Then, we can define the two following type-synonyms:

$$\begin{aligned}\mathsf{Str}^\kappa &:= \mathsf{El}(F(\mathsf{dfix}^\kappa\ F))\\ \mathsf{Str}^\kappa_\alpha &:= \mathsf{El}(\mathsf{dfix}^\kappa\ F[\alpha])\end{aligned}$$

Which give rise to the following folding and unfolding of the type of streams:

$$\begin{aligned}\mathsf{fold} :&\quad \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa_\alpha\\ \mathsf{unfold} :&\quad \mathsf{Str}^\kappa_\alpha \to \mathsf{Str}^\kappa\end{aligned}$$

By the reduction rules, we know that $\mathsf{Str}^\kappa \to^* \mathsf{Nat} \times (\ \triangleright(\alpha : \kappa).\mathsf{Str}^\kappa_\alpha)$ and so we can define

$$\begin{aligned}\mathsf{cons}^\kappa &: \mathsf{Nat} \to \triangleright^\kappa \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa\\ \mathsf{cons}^\kappa &:= \lambda(x : \mathsf{Nat}).\ \lambda(y : \triangleright^\kappa \mathsf{Str}^\kappa).\ \langle x,\ \lambda(\alpha : \kappa).\ \mathsf{fold}\ (y[\alpha])\rangle\\ \mathsf{hd}^\kappa &: \mathsf{Str}^\kappa \to \mathsf{Nat}\\ \mathsf{hd}^\kappa &:= \lambda(x : \mathsf{Str}^\kappa).\pi_1\ x\\ \mathsf{tl}^\kappa &: \mathsf{Str}^\kappa \to \triangleright^\kappa \mathsf{Str}^\kappa\\ \mathsf{tl}^\kappa &:= \lambda(x : \mathsf{Str}^\kappa).\ \lambda(\alpha : \kappa).\ \mathsf{unfold}\ ((\pi_2\ x)[\alpha])\end{aligned}$$

We can write $x ::^\kappa_\alpha xs$ for $\mathsf{cons}^\kappa\ x\ (\lambda(\alpha : \kappa).\ xs)$. Str^κ is a guarded recursive type of streams and thus all definitions of type $\mathsf{Str}^\kappa \to \mathsf{Str}^\kappa$ are guaranteed to be causal. We can use universal quantification over clocks to convert from guarded recursive types to coinductive types. Thus, $\mathsf{Str} := \forall\kappa.\mathsf{Str}^\kappa$ says that for any clock or "time-stream" κ we can make any finite observation of the stream. Another description could be a "fully constructed" stream. With the type of coinductive streams, we can lift the above definitions to also work on coinductive streams using the clock constant κ_0

$$\begin{aligned}\mathsf{cons} &: \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Str}\\ \mathsf{cons} &:= \lambda(x : \mathsf{Nat}).\ \lambda(y : \mathsf{Str}).\ \Lambda\kappa.\ x ::^\kappa_\alpha (y[\kappa])\\ \mathsf{hd} &: \mathsf{Str} \to \mathsf{Nat}\\ \mathsf{hd} &:= \lambda(x : \mathsf{Str}).\ \mathsf{hd}^{\kappa_0}(x[\kappa_0])\\ \mathsf{tl} &: \mathsf{Str} \to \mathsf{Str}\\ \mathsf{tl} &:= \lambda(x : \mathsf{Str}).\ \Lambda\kappa.\ (\mathsf{tl}^\kappa(x[\kappa]))[\diamond]\end{aligned}$$

The function nth can then be defined using standard primitive recursion over natural numbers.

$$\begin{aligned}&\mathsf{nth} : \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Nat}\\ &\mathsf{nth} = \lambda(n : \mathsf{Nat}).\ \lambda(xs : \mathsf{Str}).\ \mathsf{case}\ n\ \mathsf{of}\\ &\qquad\qquad 0 \quad\to\quad \mathsf{hd}\ xs\\ &\qquad\qquad (\mathsf{suc}\ n') \quad\to\quad \mathsf{nth}\ n'\ (\mathsf{tl}\ xs)\end{aligned}$$

3.3 Selected typing rules

Figure 11 shows some selected typing rules[1] from [4]. In the next section, we shall see how the rules are adapted (and simplified) to the simply-typed version of CloTT.

[1] Names have been added for convenience.

$$\frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda(x : A).\ t : \Pi(x : A).\ B}\ \textsc{Abs} \qquad \frac{\Gamma \vdash_\Delta t : \Pi(x : A).\ B \qquad \Gamma \vdash_\Delta u : A}{\Gamma \vdash_\Delta t\ u : B\,[u/x]}\ \textsc{App}$$

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \Lambda\kappa.\ t : \forall\kappa.\ A}\ \textsc{ClockAbs} \qquad \frac{\Gamma \vdash_\Delta t : \forall\kappa.\ A \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta t\ [\kappa'] : A\,[\kappa'/\kappa]}\ \textsc{ClockApp}$$

$$\frac{\Gamma, \alpha : \kappa \vdash_\Delta t : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).\ t : \triangleright\,(\alpha : \kappa).A}\ \textsc{TickAbs} \qquad \frac{\Gamma \vdash_\Delta t : \triangleright\,(\alpha : \kappa).A \qquad \Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta}{\Gamma, \alpha' : \kappa, \Gamma' \vdash_\Delta t\ [\alpha'] : A\,[\alpha'/\alpha]}\ \textsc{TickApp}$$

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright\,(\alpha : \kappa).A \qquad \Gamma \vdash_\Delta \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta (t\,[\kappa'/\kappa])\ [\diamond] : A\,[\kappa'/\kappa]\,[\diamond/\alpha]}\ \textsc{TConstApp} \qquad \frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A \to A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \mathsf{dfix}^\kappa\, t : \triangleright^\kappa A}\ \textsc{DFix}$$

$$\frac{\Gamma \vdash_\Delta F : \triangleright^\kappa(A \to \mathcal{U}) \to A \to \mathcal{U} \qquad \Gamma \vdash_\Delta u : A \qquad \Gamma \vdash_\Delta t : \mathsf{El}\,((\mathsf{dfix}^\kappa\, F)\ [\alpha]\ u) \qquad \Gamma \vdash_\Delta \alpha : \kappa}{\Gamma \vdash_\Delta \mathsf{unfold}_\alpha\ t : \mathsf{El}\,(F\ (\mathsf{dfix}^\kappa\, F)\ u)}\ \textsc{Unfold}$$

$$\frac{\Gamma \vdash_\Delta F : \triangleright^\kappa(A \to \mathcal{U}) \to A \to \mathcal{U} \qquad \Gamma \vdash_\Delta u : A \qquad \Gamma \vdash_\Delta t : \mathsf{El}\,(F\ (\mathsf{dfix}^\kappa\, F)\ u) \qquad \Gamma \vdash_\Delta \alpha : \kappa}{\Gamma \vdash_\Delta \mathsf{fold}_\alpha\ t : \mathsf{El}\,((\mathsf{dfix}^\kappa\, F)\ [\alpha]\ u)}\ \textsc{Fold}$$

Figure 11: Selected typing rules from [4].

As in all dependently typed calculi, normal arrow types are simply modeled with Π-types that do not have the bound variable free in their codomain. There is a single universe $\mathcal{U}$ in this presentation of CloTT and the meta-function El gives the type that corresponds to a "code" in the universe $\mathcal{U}$. The rules Fold and Unfold show how recursive types are treated using the dfix primitive.

Figure 12 show the typing rules for the simply-typed fragment of CloTT. Standard typing rules for e.g. sums and products are left out. Note that they are somewhat simpler (unsurprisingly). The $\triangleright\,(\alpha : \kappa).$ term is gone, since types can no longer depend on values, and therefore they cannot depend on ticks either. Normal application and clock application also no longer substitutes in the conclusion type, again as a direct consequence of simple types. The rule for applying the tick-constant is remarkably simpler. The side-condition that Γ should be κ-stable remains, but otherwise it simply removes the guardedness from the type of a term. We do not have to substitute in neither t nor A, nor guess a κ' to use. This shows another subtle consequence of being simply-typed - in a dependently typed setting, one typically normalizes terms under binders, since syntactical equality is often very interesting. In a simply-typed setting, this is far less common and less useful, and eliminating this behaviour helps in simplifying this rule substantially.

Recursive types no longer arise from the fixpoint combinator, but are modeled separately using the Fix type constructor, along with new fold and unfold combinators that are substially simpler than their dependently typed counterparts. Notice that neither the Fix type or its (un)foldings directly mention any clock variables - it is simply through the Fix rule that their guarded nature arises, since the premise requires t to have type $\triangleright^\kappa A \to A$. The formation rules for recursive types is predicated on the well-formedness of $\mathsf{Fix}\,X.A$; namely that X can only occur strictly positively (guarded or unguarded in theory), or, if X occurs negatively it must be guarded by a $\triangleright^\kappa$. Note that this calculus does *not* have the delayed fixpoint from CloTT. fix^κ type-checks as A and not $\triangleright^\kappa A$. Intuitively, fix^κ could be defined simply as $(\mathsf{dfix}^\kappa\, t)[\diamond]$, but the delayed fixpoint is much less useful in a simply-typed setting, so we have omitted it entirely here.

$$\frac{\Gamma, x : A, \Gamma' \vdash_\Delta}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda(x : A).\ t : A \to B}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash_\Delta t : A \to B \qquad \Gamma \vdash_\Delta s : A}{\Gamma \vdash_\Delta t\, s : B}\ \textsc{App} \qquad \frac{\Gamma, \alpha : \kappa \vdash_\Delta t : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).\ t : \triangleright^\kappa A}\ \textsc{TickAbs}$$

$$\frac{\Gamma \vdash_{\Delta,\kappa} t : A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \Lambda\kappa.\ t : \forall\kappa.\ A}\ \textsc{ClockAbs} \qquad \frac{\Gamma \vdash_\Delta t : \forall\kappa.\ A \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta t\ [\kappa'] : A\,[\kappa'/\kappa]}\ \textsc{ClockApp}$$

$$\frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A \qquad \Gamma' \vdash_\Delta}{\Gamma, \alpha : \kappa, \Gamma' \vdash_\Delta t\ [\alpha] : A}\ \textsc{TickApp} \qquad \frac{\Gamma \vdash_{\Delta,\kappa} t : \triangleright^\kappa A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_{\Delta,\kappa} t\ [\diamond] : A}\ \textsc{TConstApp}$$

$$\frac{\Gamma \vdash_\Delta t : \triangleright^\kappa A \to A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \mathsf{fix}^\kappa t : A}\ \textsc{Fix} \qquad \frac{\Gamma \vdash_\Delta t : \mathsf{Fix}\, X.A \qquad \kappa \in \Delta \qquad \Gamma \vdash \mathsf{Fix}\, X.A}{\Gamma \vdash_\Delta \mathsf{unfold}\, t : A\,[\mathsf{Fix}\, X.A/X]}\ \textsc{Unfold}$$

$$\frac{\Gamma \vdash_\Delta t : A\,[\mathsf{Fix}\, X.A/X] \qquad \kappa \in \Delta \qquad \Gamma \vdash \mathsf{Fix}\, X.A}{\Gamma \vdash_\Delta \mathsf{fold}\, t : \mathsf{Fix}\, X.A}\ \textsc{Fold}$$

Figure 12: Selected typing rules from simply-typed CloTT.

While 12 shows the structure of the core calculus, the typing rules alone (even with the omitted standard rules for sums and products) are a far cry from a specification of a practically useful language.

4 CloFRP

We shall now concentrate on designing a practically useful language for FRP based on the rules of simply-typed CloTT. For convenience, we shall name this language CloFRP for *Clocked Functional Reactive Programming*.

To reach our goal of a practically useful language, there are certain things we must add to the underlying calculus to make the authoring of non-trivial programs feasible. Furthermore, we must alter the syntax to make it more natural to write programs on a standard ASCII keyboard.

4.1 Concrete syntax

The concrete syntax is mostly similar to the presentation of the calculus, but substitutes some mathematical symbols with their ASCII counterparts. Furthermore, it introduces some additional syntax to disambiguate parsing. Otherwise, the syntax is very inspired by Haskell, but is not white-space sensitive, and therefore takes additional inspiration from Coq and ML. Specifically

- $\lambda(x : A).\ t$ becomes `\(x : A) -> t`,
- $\lambda(\alpha : \kappa).\ t$ becomes `\\(a : k) -> t`,
- $\forall a.A$ becomes `forall a. A`,
- $\diamond$ becomes `<>`,
- $\triangleright^\kappa$ becomes `|>k`,
- case expresssions are written `case e of | p₁ -> e₁ | p₂ -> e₂ | ... | pₙ -> eₙ`, that is, clauses are delimited by pipes,

- a program consists of several declarations, and each declaration *must* be terminated by a full-stop (“.”).
- Clock application is disambiguated by using braces instead of brackets. In fact, clock application is no longer a stand-alone concept, but we will get to that later.

4.2 Features

Parametric polymorphism is necessary to avoid rampant duplication of code in a statically typed functional setting. Parametric polymorphism allows definitions and data-types to quantify over types, thereby making them far more general-purpose and useful. The canonical example is the identity function `(\x -> x) : forall a. a -> a`. Note that the original formulation of CloTT does not support polymorphism in general - only clock quantification is allowed.

Algebraic Data Types allow programmers to describe their own data-types as sums of named product-type constructors. They are sometimes simply called data-type declarations. In Haskell, an example of such an ADT declaration could be

```
data Maybe a = Nothing | Just a.
```

Such a declaration says a few things:

- `Maybe` is a type-constructor (sometimes called a type-operator) that takes an inhabited type and returns an inhabited type. A formal way to say this is to say that the kind of `Maybe` is `* -> *` which is often just written `Maybe : * -> *`.
- There are two ways to construct a value of type `Maybe a`: by using the `Nothing` constructor, or the `Just` constructor. The `Just` constructor is unary and expects to be given an argument of type `a`. Another way to say this is that the declaration above gives rise to two constructors, namely `Nothing : forall (a : *). Maybe a` and `Just : forall (a : *). a -> Maybe a`.
- Given any value `x` of type `Maybe a`, we can inspect `x` by pattern matching on it. Thus, `x` can have two forms: either `x = Nothing` or `∃ (x' : a). x = Just x'`

ADTs in this simple form makes it far more ergonomic to express complicated types and operations on their values while programming. Note that the fact that ADTs can express type-constructors, which along with parametric polymorphism makes our language as expressive as $\mathsf{SystemF}_\omega$ [34].

Pattern matching is connected to ADTs and facilitate a way to de-construct and inspect values of a certain type. Modern programming languages feature nested pattern matching, so that one could for example implement `forall (a : *). a -> Maybe (Maybe a) -> a` as

```
\default x ->
  case x of
  | Just (Just x') -> x'
  | _              -> default
```

Just like in Haskell, one can also pattern match in let-bindings. Note that the current implementation does not do coverage checking, so pattern matches can easily be incomplete.

Type-synonyms are a way to give a synonym for a type expression. Synonyms are mostly a convenience feature. They can be used as a simple textual substition to improve the self-documenting nature of code, as in `type Name = String`, or they can be parameterised which induces a limited form of type-level abstraction, as in `type Pair a b = (a,b)`. Type-synonyms can only be fully-applied, and can mention other type-synonyms but cannot be recursive in any way. Type-synonyms are resolved as part of the elaboration step, so type-checking is completely ignorant of them.

Functor deriving allows the programmer to explicitly tell the compiler to derive the definition of the `fmap` function, which is used to map over type-constructors that are functorial. Haskell has pioneered an advanced system for deriving instances of type-classes that can save programmers significant work. However, our language will not support type-classes at this time, and only one form of hard-coded deriving of functors. The motivation for this feature lies mostly in the implementation, since we need functoriality to perform primitive recursion over recursive types. Concretely, one can declare an ADT to derive `fmap` by writing e.g. `data Maybe a = Nothing | Just a deriving Functor`.

4.3 An example program

We shall later go into more detail about more advanced programs in CloFRP, but just to give an idea of the language we will be describing, we show a small example here; Listing 1 encodes in CloFRP the same example we saw in CloTT in Section 3.2.

```
1  data StreamF (k : Clock) a f = Cons a (|>k f) deriving Functor.
2  type Stream  (k : Clock) a   = Fix (StreamF k a).
3  data CoStream a              = Cos (forall (k : Clock). Stream k a).
4  data NatF f                  = Z | S f deriving Functor.
5  type Nat                     = Fix NatF.
6
7  uncos : forall (k : Clock) a. CoStream a -> Stream k a.
8  uncos = \xs -> case xs of | Cos xs' -> xs'.
9
10 consk : forall (k : Clock) a. a -> |>k (Stream k a) -> Stream k a.
11 consk = \x xs -> fold (Cons x xs).
12
13 cons : forall (k : Clock) a. a -> |>k (CoStream a) -> CoStream a.
14 cons = \x xs ->
15   Cos (fold (Cons x (\\(af : k) -> uncos (xs [af])))).
16
17 hdk : forall (k : Clock) a. Stream k a -> a.
18 hdk = \xs ->
19   case unfold xs of
20   | Cons x xs' -> x.
21
22 tlk : forall (k : Clock) a. Stream k a -> |>k (Stream k a).
23 tlk = \xs ->
24   case unfold xs of
25   | Cons x xs' -> xs'.
26
27 hd : forall a. CoStream a -> a.
28 hd = \xs -> hdk {K0} (uncos xs).
29
30 tl : forall a. CoStream a -> CoStream a.
31 tl = \xs -> Cos (tlk (uncos xs) [<>]).
32
33 nth : Nat -> CoStream Nat -> Nat.
34 nth = \n xs ->
35   let Cos xs' = xs in
36   let fn = \n xs' ->
37     case n of
38     | Z -> hdk xs'
39     | S (n', r) -> r (tlk xs' [<>])
40   in  primRec {NatF} fn n xs'.
```

Listing 1: The CloTT example from 3.2 encoded in CloFRP.

Here `Stream Nat` corresponds to Str^{κ} and `CoStream Nat` corresponds to coinductive streams Str in CloTT. We have to wrap the forall-quantification of coinductive streams in a data-type to make sure the inference engine will not instantiate the quantifier too early, like we would have to in Haskell.

5 Inference system for CloFRP

Since CloFRP is designed with practical usage in mind, the inference system is a core part of its specification. The actual inference algorithm is based on [30], and we shall therefore present a brief review of the paper, before presenting our own derived inference system (in a high-level declarative and algorithmic form).

5.1 Complete and easy bidirectional type-inference

Our inference system is based upon the algorithm described in "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism" [30] by Joshua Dunfield and Neel Krishnaswami. In order to present our inference rules, an understanding of their algorithm is required. Therefore, we shall provide a brief exposition of their paper.

The main contribution of [30] is an inference algorithm that handles higher-rank predicative polymorphism, but which also dispenses with complex elaboration steps from source code into a typeable representation. Rather, one can typecheck an AST corresponding quite naturally to the concrete syntax of a typical functional language. Concretely, their algorithm is a quite standard bidirectional inference algorithm, which has a type-checking and type-synthesizing judgment. They add a third judgment, the so called "application-synthesis" judgment, which deals with inferring type-arguments to polymorphic functions specifically. In addition, they model type-instantiation (instantiation a polymorphic quantifier with a type) by defining a subtyping relation, where subtyping is defined as a "more-polymorphic-than" relation that guesses type instantiatings arbitrary deeply with types. The result is a type-inference system that handles higher-rank types with quantifiers in arbitrary positions, and which retains the η-law (namely that if $(\lambda x.\ f\,x) : A$ and $x \notin FV(f)$, then $f : A$).

The paper contains two formulations of their inference system: a declarative system, which is easier to understand but employs guessing and thus does not directly yield an algorithm, and an algorithmic system that is somewhat more complex but from which one can derive an implementation. They prove that their algorithmic system is sound and complete with respect to their declarative specification of type-inference.

5.1.1 Declarative system

The declarative system consists of three main judgments, presented in Figure 13, and two auxiliary judgments namely context well-formedness, and subtyping, shown in Figure 14. The syntax describes declarative contexts as Ψ, while x are names, e_i are expressions and A, B, C are possibly polymorphic types while τ, σ are monomorphic types. Type-variables are denoted by α, β.

$\boxed{\Psi \vdash e \Leftarrow A}$ Under context Ψ, e checks against type A

$\boxed{\Psi \vdash e \Rightarrow A}$ Under context Ψ, e synthesizes type A

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C}$ Under context Ψ, applying a function of type A to e synthesizes type C

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \textsc{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A \qquad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B}\ \textsc{DeclSub}$$

$$\frac{\Psi \vdash A \qquad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \textsc{DeclAnno} \qquad \frac{}{\Psi \vdash () \Leftarrow 1}\ \textsc{Decl1I} \qquad \frac{}{\Psi \vdash () \Rightarrow 1}\ \textsc{Decl1I}\Rightarrow$$

$$\frac{\Psi, \alpha \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall \alpha.\ A}\ \textsc{Decl}\forall\textsc{I} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash [\tau/\alpha]A \bullet e \Rrightarrow C}{\Psi \vdash \forall \alpha.\ A \bullet e \Rrightarrow C}\ \textsc{Decl}\forall\textsc{App}$$

$$\frac{\Psi, x : A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\, e \Leftarrow A \to B}\ \textsc{Decl}{\to}\textsc{I} \qquad \frac{\Psi \vdash \sigma \to \tau \qquad \Psi, x : \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\, e \Rightarrow \sigma \to \tau}\ \textsc{Decl}{\to}\textsc{I}\Rightarrow$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \qquad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\, e_2 \Rightarrow C}\ \textsc{Decl}{\to}\textsc{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash A \to C \bullet e \Rrightarrow C}\ \textsc{Decl}{\to}\textsc{App}$$

Figure 13: The declarative system from [30].

Most of the rules are pretty standard bidirectional type-inference rules: DeclVar says that variables infer the types they are assigned in the context, and DeclSub expresses the rule of subsumption – an expression checks against a type B if the type it synthesizes is a subtype of B. As mentioned above, subtyping here means "more-polymorphic-than". For example, if $\mathsf{id} \Rightarrow \forall a.\, a \to a$ then $\mathsf{id} \Leftarrow 1 \to 1$. Dually, DeclAnno switches from synthesis to checking, by synthesizing A for e if e checks against the annotated type A. Decl$\forall$I introduces universally quantified types, and Decl$\to$I checks a lambda against an arrow type. Less standard is the Decl$\to$I rule, which synthesizes a monomorphic type for a lambda. Application of terms (Decl$\to$E) switches to the application-synthesis judgment. This rule says that $e_1\, e_2$ synthesizes C if e_1 synthesizes A and a function of type A applied to e_2 synthesizes C. The application-synthesis judgment proceeds in two ways depending on the form of A. If A is polymorphic, then Decl$\forall$App guesses a monotype τ for the universally quantified variable α, and substitutes τ for α before proceeding inductively. If A is a function type yielding a C, it simply checks e against A before yielding C

$\boxed{\Psi \vdash A}$ Under context Ψ, type A is well-formed

$\boxed{\Psi \vdash A \leq B}$ Under context Ψ, type A is a subtype of B

$$\frac{\alpha \in \Psi}{\Psi \vdash a}\ \textsc{DeclUvarWF} \qquad \frac{}{\Psi \vdash 1}\ \textsc{DeclUnitWF} \qquad \frac{\Psi \vdash A \qquad \Psi \vdash B}{\Psi \vdash A \to B}\ \textsc{DeclArrowWF}$$

$$\frac{\Psi, \alpha \vdash A}{\Psi \vdash \forall \alpha.\, A}\ \textsc{DeclForallWF} \qquad \frac{\alpha \in \Psi}{\Psi \vdash \alpha \leq \alpha}\ \leq\textsc{Var} \qquad \frac{}{\Psi \vdash 1 \leq 1}\ \leq\textsc{Unit}$$

$$\frac{\Psi \vdash B_1 \leq A_1 \qquad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \leq{\to} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash [\tau/\alpha]A \leq B}{\Psi \vdash \forall \alpha.\, A \leq B}\ \leq\forall\mathrm{L}$$

$$\frac{\Psi, \beta \vdash A \leq B}{\Psi \vdash A \leq \forall \beta.\, B}\ \leq\forall\mathrm{R}$$

Figure 14: Declarative well-formedness and subtyping rules from [30].

The well-formedness relation in Figure 14 is completely standard. Subtyping also proceeds as expected, with a contravariant twist for the function type. $\leq\forall$L guesses a type τ out of thin air, which is one of the reasons this is only a declarative specification. $\leq\forall$R is slightly subtler – A is a subtype of $\forall\beta.\, B$ if $A \leq B$ in a context extended with the universal variable β. The intution is that since $\forall\beta.\, B$ is a subtype of $[\tau/\beta]B$ for any arbitrary τ, then A must also subtype $[\tau/\beta]B$ for any τ. That means that if we can show that $A \leq B$ with a free variable β, then we should be able to substitute any τ for β in B and have that $A \leq [\tau/\beta]B$.

While the declarative system briefly outlined above is a fine specification of how type-inference should behave, it employs guessing far too much – Decl$\forall$App and Decl$\to$I cannot be implemented without an oracle, and the same applies to the $\leq\forall L$ subtyping rule. To deal with these issues, [30] develop an algorithmic version of the declarative rules – this new system gives rise to an implementation, but is significantly more complicated.

5.1.2 Algorithmic system

The algorithmic system sets out to fix the three oracular rules by deferring the choice of the type to a later time. It does so by introducing *existential variables* that represent some type "we do not know yet". They're not quite unification variables as they are organized into *ordered algorithmic contexts*. The order in such a context allows the system to carefully control the scope of existential variables and the free variables in their solutions.

The algorithmic system consists of subtyping rules (Figure 16), instantiation rules (Figure 17) and typing rules (Figure 18).

The syntax is mostly identical to the declarative system, but types and monotypes now also contain existential variables $\hat{\alpha}$.

Algorithmic contexts are defined by the grammar $\Gamma, \Delta, \Theta ::= \cdot \mid \Gamma, \alpha \mid \Gamma, x : A \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}}$. Contexts contain universal variables, assumptions that a name has a type, unsolved existentials, solved existentials, and *scope markers* $\blacktriangleright_{\hat{\alpha}}$, which we shall explain in more detail later.

The order of elements in the contexts is important – all free variables of a type must be in the context *before* the element. For example, $(\alpha, x : 1 \to \alpha)$ is well-formed, whereas $(x : 1 \to \alpha, \alpha)$ is not. The same principle applies to solved existentials. This well-formedness judgment is formalized in [30], but we will omit it here for brevity.

Contexts can be applied as a substitution to a type, written $[\Gamma]A$. This will substitute all existentials in A with their solutions in Γ. This operation is formalized in Figure 15. Contexts can not only be manipulated by appending to the right – one can also insert and replace in the middle of contexts. $\Gamma[\hat{\alpha}]$ means that Γ contains $\hat{\alpha}$ somewhere, and consequently the syntax $\Gamma[\hat{\alpha} = \tau]$ solves $\hat{\alpha}$ to τ "in-place". For example, if we have $\Gamma[\hat{\beta}] = (\hat{\alpha}, \hat{\beta}, x : \hat{\beta})$ then $\Gamma[\hat{\beta} = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta})$.

$$
\begin{array}{lcl}
[\Gamma]\alpha & = & \alpha \\
[\Gamma]1 & = & 1 \\
[\Gamma[\hat{\alpha} = \tau]]\hat{\alpha} & = & [\Gamma[\hat{\alpha} = \tau]]\tau \\
[\Gamma[\hat{\alpha}]]\hat{\alpha} & = & \hat{\alpha} \\
[\Gamma](A \to B) & = & ([\Gamma]A) \to ([\Gamma]B) \\
[\Gamma](\forall\alpha.\, A) & = & \forall\alpha.\, [\Gamma]A
\end{array}
$$

Figure 15: Context substitution as defined in [30].

In contrast to the declarative rules, all the judgments in the algorithmic system yield an *output context*. That is, application of the rules evolves the contexts, and output contexts can be the input context of other premises. Unsolved existentials get solved when they are compared against a type; for example

$\hat{\alpha} <: \beta$ leads to replacing the unsolved existential $\hat{\alpha}$ with the solution $\hat{\alpha} = \beta$ in the output context. Thus, input contexts evolve into output contexts that are "more solved".

The most prominent changes from the declarative to the algorithmic system is in the subtyping rules, so we shall elaborate on those first.

$\boxed{\Gamma \vdash A <: B \dashv \Delta}$ Under input context Γ, type A is a subtype of B, with output context Δ

$$\frac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\ {<:}\textsc{Var} \qquad \frac{}{\Gamma \vdash 1 <: 1 \dashv \Gamma}\ {<:}\textsc{Unit} \qquad \frac{}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}\ {<:}\textsc{Exvar}$$

$$\frac{\Gamma \vdash B_1 <: A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Delta}\ {<:}{\to} \qquad \frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash \forall \alpha.\, A <: B \dashv \Delta}\ {<:}\forall\mathrm{L}$$

$$\frac{\Gamma, \alpha \vdash A <: B \dashv \Delta, \alpha, \Theta}{\Gamma \vdash A <: \forall \alpha.\, B \dashv \Delta}\ {<:}\forall\mathrm{R} \qquad \frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: A \dashv \Delta}\ {<:}\textsc{InstantiateL}$$

$$\frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}] \vdash A \leqq: \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A <: \hat{\alpha} \dashv \Delta}\ {<:}\textsc{InstantiateR}$$

Figure 16: Algorithmic subtyping rules from [30].

The subtyping rules are almost identical to their declarative counterparts. Rule $<:$Exvar is simply reflexivity for existentials. Rule $<:\to$ is more interesting. The output context of the first premise is applied as a substitution to the arguments of the second premise. This is a pattern that occurs in many of the other rules: whenever we use a rule we shall always fully apply the arguments under the current context. This maintains an invariant that the types never contain solved existentials in the current context, which simplifies the system. Rule $<:\forall$L diverges substantially from its declarative formulation. Here, we introduce a new existential $\hat{\alpha}$ for the quantified variable α. We add a *scope marker* $\blacktriangleright_{\hat{\alpha}}$ to the context so we can control the scope of the introduced existential. Consequently, the output context only retains context elements to the left of the marker. Crucially, $\hat{\alpha}$ could have been solved by the premise $[\hat{\alpha}/\alpha]A <: B$, so we cannot simply use $\hat{\alpha}$ to indicate scope. Additionally, more existentials could have been added to the left of $\hat{\alpha}$, in particular by *articulation* (which we will see later), which articulates $\hat{\alpha}$ to $\hat{\alpha}_1 \to \hat{\alpha}_2$. Since $\hat{\alpha}$ could mention these, and $\hat{\alpha}$ goes out of scope after the rule, we must drop these from the output context as well. The $<:\forall R$ is close to the corresponding declarative rule, but also drops the trailing context Θ for scoping reasons. It does not need an explicit marker since the universal variable α cannot be articulated and so can act as its own scoping marker. The last two rules are especially important as they derive subtypings for an existential variable and a type (on either side). They just check for circularity and then offload their work to the instantiation rules.

$\boxed{\Gamma \vdash \hat{\alpha} :\leqq A \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $\hat{\alpha} <: A$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \hat{\alpha} :\leqq \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \textsc{InstLSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} :\leqq \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \textsc{InstLReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash A_1 \leqq: \hat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 :\leqq [\Theta]A_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A_1 \to A_2 \dashv \Delta}\ \textsc{InstLArr}$$

$$\frac{\Gamma[\hat{\alpha}], \beta \vdash \hat{\alpha} :\leqq B \dashv \Delta, \beta, \Delta'}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq \forall \beta.\ B \dashv \Delta}\ \textsc{InstLAllR}$$

$\boxed{\Gamma \vdash A \leqq: \hat{\alpha} \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $A <: \hat{\alpha}$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \tau \leqq: \hat{\alpha} \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \textsc{InstRSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\beta} \leqq: \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \textsc{InstRReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \hat{\alpha}_1 :\leqq A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \leqq: \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A_1 \to A_2 \leqq: \hat{\alpha} \dashv \Delta}\ \textsc{InstRArr}$$

$$\frac{\Gamma[\hat{\alpha}], \blacktriangleright_{\hat{\beta}}, \hat{\beta} \vdash [\hat{\beta}/\beta]B \leqq: \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{\beta}}, \Delta'}{\Gamma[\hat{\alpha}] \vdash \forall \beta.\ B \leqq: \alpha \dashv \Delta}\ \textsc{InstRAllL}$$

Figure 17: Algorithmic instantantiation rules from [30].

Instantiation is defined by two almost-symmetric rules: one instantiates the existential $\hat{\alpha}$ to a subtype of A while the other instantiates $\hat{\alpha}$ to a supertype of A. These two rules both yield an output context that is at least as solved as the input context. In the case where A is a mono-type, then instantiation is guaranteed to result in a strictly more solved output context.

InstLSolve simply solves an existential to a mono-type. InstLReach is a bit more subtle; if attempting to instantiate $\hat{\alpha}$ to $\hat{\beta}$ in a context where $\hat{\alpha}$ comes before $\hat{\beta}$, we switch the solution around so $\hat{\beta} = \hat{\alpha}$, since the naïve approach would make the context non-well-formed. InstLArr is also a bit complicated. Intuitively, the task of solving an existential $\hat{\alpha}$ to a function type $A_1 \to A_2$ is really the task of articulating $\hat{\alpha}$ to $\hat{\alpha}_1 \to \hat{\alpha}_2$, solving $\hat{\alpha}_1$ to a supertype of A_1 (contravariance at play again) and solving $\hat{\alpha}_2$ to a subtype of A_2. Again, we make sure to substitute away all solved existentials in the second premise. Finally, InstLAllR maintains predicativity by decomposing the quantifier as in the subtyping rules. The right-instantiation rules are symmetrical to the left-instantiation rules, except for InstRAllL which is the instantiation version of $<:\forall$L.

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsc{Sub}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow A \dashv \Delta}\ \textsc{Anno} \qquad \frac{}{\Gamma \vdash () \Leftarrow 1 \dashv \Gamma}\ 1\mathrm{I} \qquad \frac{}{\Gamma \vdash () \Rightarrow 1 \dashv \Gamma}\ \mathrm{I}\Rightarrow$$

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall \alpha.\ A \dashv \Delta}\ \forall\mathrm{I} \qquad \frac{\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha.\ A \bullet e \Rrightarrow C \dashv \Delta}\ \forall\textsc{App}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ {\to}\mathrm{I} \qquad \frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta}\ {\to}\mathrm{I}\Rightarrow$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow C \dashv \Delta}\ {\to}\mathrm{E} \qquad \frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2 \dashv \Delta}\ \hat{\alpha}\textsc{App}$$

$$\frac{\Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Rrightarrow C \dashv \Delta}\ {\to}\textsc{App}$$

Figure 18: Algorithmic typing rules from [30].

The algorithmic typing rules are quite similar to the declarative rules, but with extra context-handling logistics. Subsumption (Sub) takes care to maintain fully applied types in its second premise, which switches to the subtyping judgment. $\forall$I adds a quantified variable to the context and derives $e \Leftarrow A$, but drops α and any context-elements that depend on α (and thus live in Θ) in the output contex. The $\forall$App rule is in many ways similar to $<:\forall$L, but does *not* place a marker before $\hat{\alpha}$; $\hat{\alpha}$ may appear in the result type C so it must survive in the output context Δ. $\to$I is similar to the declarative rule, whereas $\to$I$\Rightarrow$ is more interesting, as it corresponds to one of the declarative oracular rules. Instead of guessing types, it generates fresh existentials $\hat{\alpha}, \hat{\beta}$ and derives $e \Leftarrow \hat{\beta}$ in a context extended with $\hat{\alpha}, \hat{\beta}, x : \hat{\alpha}$. Again, no marker is placed as the fresh existentials appear in the output type. Rule $\to$E is the analogue of Decl$\to$E and switches to the application synthesis judgment. $\hat{\alpha}$App is the only "new" rule; it encodes the constraint that if applying a function of some existential type $\hat{\alpha}$ to e, then we know that $\hat{\alpha}$ must have the form of an arrow type. Finally, $\to$App is analogous to its declarative rule Decl$\to$App.

The example in Figure 19 shows how to derive an application of the the identity function $\mathsf{id} : \forall a.a \to a$ to the unit constructor ().

Let $\Gamma = \cdot, \mathsf{id} : (\forall a.a \to a)$

$$\dfrac{\dfrac{}{\Gamma \vdash \mathsf{id} \Rightarrow (\forall a.a \to a) \dashv \Gamma}\,\textsc{Var} \qquad \dfrac{\dfrac{\dfrac{\Gamma, \hat{\alpha} \vdash () \Rightarrow 1 \dashv \Gamma, \hat{\alpha} \qquad T_3}{\Gamma, \hat{\alpha} \vdash () \Leftarrow \hat{\alpha} \vdash \Gamma, \hat{\alpha} = 1}\,\textsc{Sub}}{\Gamma, \hat{\alpha} \vdash \hat{\alpha} \to \hat{\alpha} \bullet () \Rrightarrow \hat{\alpha} \dashv \Gamma, \hat{\alpha} = 1}\,{\to}\textsc{App}}{\Gamma \vdash (\forall a.a \to a) \bullet () \Rrightarrow \hat{\alpha} \dashv \Gamma, \hat{\alpha} = 1}\,\forall\textsc{App}}{\dfrac{\Gamma \vdash \mathsf{id}\ () \Rightarrow \hat{\alpha} \dashv \Gamma, \hat{\alpha} = 1}{\Gamma \vdash \mathsf{id}\ () \Leftarrow 1 \dashv \Gamma, \hat{\alpha} = 1}\,\textsc{Sub}}\,{\to}\textsc{E} \qquad T_2$$

$$T_2 = \dfrac{}{\Gamma, \hat{\alpha} = 1 \vdash 1 <: 1 \dashv \Gamma, \hat{\alpha} = 1}\,{<:}\textsc{Unit}$$

$$T_3 = \dfrac{\dfrac{}{\Gamma, \hat{\alpha} \vdash 1 =<: \hat{\alpha} \dashv \Gamma, \hat{\alpha} = 1}\,\textsc{InstRSolve}}{\Gamma, \hat{\alpha} \vdash 1 <: \alpha \dashv \Gamma, \hat{\alpha} = 1}\,\textsc{InstantiateR}$$

Figure 19: An example of deriving a polymorphic application.

Notice that in the root of T_2, we are deriving $1 <: 1$ because we have applied the output context of the first premise (namely $\Gamma, \hat{\alpha} = 1$) to the inferred type $\hat{\alpha}$, which results in $[\Gamma, \hat{\alpha} = 1]\hat{\alpha} \equiv 1$.

This concludes this brief review of the inference system presented in [30]. The reader is adviced to consult the original paper for additional details. As previously mentioned, the inference system of CloFRP can be considered an extension of the system presented in [30]. We shall therefore follow their schema and present our extension in first a high-level declarative form, and later in a more detailed algorithmic form.

5.2 Declarative inference for CloFRP

The main additions in CloFRP (when considering type-inference) compared to the simple language used in [30] are pattern matching, clocks and ticks on them, recursion and recursive types, and algebraic data-type declarations. Especially the last addition mandates a fundamental change in the presentation of the calculus, since we can no longer assume a finite set of types and their constructors, but must accomodate an open set instead. Furthermore, we must consider type-constructors, that is, types that depend on types.

5.2.1 Syntax

The syntax of CloFRP can be seen in Figure 20, specified as an EBNF grammar.

$$
\begin{array}{llll}
\text{Kinds} & \chi & ::= & \star \mid \mathsf{c} \mid \chi \to \chi' \\
 & \chi' & ::= & \star \mid \chi \to \chi' \\
\text{Types} & A, B, C, F, G & ::= & \mathcal{T} \mid A \to B \mid \langle A_1, \ldots, A_n \rangle \mid \alpha \mid \forall \alpha : \chi.\, A \mid F\, B \mid \mathsf{Fix}\, F \mid \triangleright^{\kappa} A \\
\text{Monotypes} & \tau, \sigma & ::= & \mathcal{T} \mid \tau \to \sigma \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \tau\, \sigma \mid \alpha \mid \mathsf{Fix}\, \tau \mid \triangleright^{\kappa} \tau \\
\text{Terms} & e & ::= & x \mid \mathcal{C} \mid \lambda x.e \mid e_1\, e_2 \mid \langle e_1, \ldots, e_n \rangle \mid (e : A) \mid \gamma(x : \kappa).\, e \mid e\,[x] \mid e\,\{A\} \mid \mathsf{fix} \\
 & & & \mid \mathsf{fold} \mid \mathsf{unfold} \mid \mathsf{fmap}_F \mid \mathsf{primRec}_F \\
 & & & \mid \mathsf{case}\ e\ \mathsf{of}\ (\rho_1 \longrightarrow e_1, \ldots, \rho_n \longrightarrow e_n) \mid \mathsf{let}\, \rho\ = e_1\, \mathsf{in}\, e_2 \\
\text{Patterns} & \rho & ::= & x \mid \mathcal{C}\, \rho_1, \ldots, \rho_n \mid \langle \rho_1, \ldots, \rho_n \rangle \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, x :_{\lambda} A \mid \Psi, x :_{=} A \mid \Psi, \mathcal{C} : A \mid \Psi, \alpha : \chi \mid \Psi, \mathcal{T} : \chi \mid \mathit{Functor}(F)
\end{array}
$$

Figure 20: The syntax of the declarative specification of CloFRP.

Kinds are either the inhabited kind $\star$, the kind of clocks c, or the kind of type-constructors $\chi \to \chi'$. χ' doesn't include the kind of clocks, since we cannot have clocks that are indexed over anything. Thus, a clock-kind directly to the right of an arrow is not valid syntax. $\mathsf{c} \to \star$ is valid, as is $(\mathsf{c} \to \star) \to \star$, but $\star \to \mathsf{c}$ or $(\star \to \mathsf{c}) \to \star$ is not.

Possibly polymorphic types can be the name of a declared type $\mathcal{T}$ in the set of declared types. Examples could be Nat or Unit. We loosely use the symbols A, B, C to refer to types of kind $\star$, and F, G to refer to type constructors of kind $\chi \to \chi$. We have the standard function space, and n-ary tuples of types. α signifies the use of a quantified type-variable, and are thus not related to ticks anymore, to avoid ambiguity. Universal quantification of types mentions the kind of the quantified type in $\forall \alpha : \chi.\, A$. We shall often omit the kind-signature when we simply mean $\alpha : \star$. Types can be applied to other types, written $F\, B$. We choose to represent recursive types as a built-in type-constructor Fix. Finally, we have the type of delayed computations on clock κ written $\triangleright^{\kappa} A$. Crucially, clock variables are just normal type variables of kind c, and so do not have any special syntatic form. **Monotypes** are just normal types without quantification.

Terms are variables, constructor names $\mathcal{C}$ (e.g. Nil or Cons), lambda abstractions, term applications, n-ary tuples, annotated terms, tick abstractions $\gamma(x : \kappa).\, e$, tick application $e\,[x]$, and explicit type-applications $e\,\{A\}$. We use new syntax for tick-abstractions to avoid ambiguity, since the form that CloTT uses – $\lambda(\alpha : \kappa).\, t$ – is too easy to confuse with a normal lambda, and α is used for polymorphism instead. γ looks kind of like an upside-down lambda, so the symbol does signify some relation with the orignal syntax.

Fixpoints fix are the guarded recursive fixpoints of simply-typed CloTT. The normal fixpoint has type $\mathsf{fix} : (A \to A) \to A$, whereas the guarded fixpoint has type $\mathsf{fix} : (\triangleright^{\kappa} A \to A) \to A$. The guarded fixpoint allows a programmer to encode productive and causal coinductive definitions in a rather elegant manner. This is perfectly fine for the specific domain of FRP, but for other domains non-causal recursion can be useful. This is exactly what CloTT, and by extension CloFRP, aims to solve through clock quantification.

We can fold and unfold recursive types and map over functorial type-constructors F with fmap_F.

We can write primitive recursive definitions over functorial type-constructors as well, in the same sense as in [3, 14], using $\mathsf{primRec}_F$. A primitive recursion combinator allows us to express well-founded inductive definitions without a sophisticated termination checker. The downside is that it is somewhat clunky to work with, and that a considerable class of more complicated terminating recursive functions cannot be encoded.

Finally, there are case expressions with lists of branches and let-expressions. **Branches** are simply a pattern and an expression.

Patterns are simply name-bindings x, constructor-patterns $\mathcal{P}$ applied to zero or more patterns, or n-ary tuple patterns $\langle \rho_1, \ldots, \rho_n \rangle$.

Declarative contexts are unordered bags that carry contextual information used for typing. Names can be associated to types in two ways: through a lambda abstraction as $x :_\lambda A$, or through a let-binding or pattern-match as $x :_= A$. Constructors are bound to types as $\mathcal{C} : A$, and declared types are associated with a kind as $\mathcal{T} : \chi$, while type-variables are associated as $\alpha : \chi$. Finally, we can assert that a type F of kind $\star \to \star$ is functorial as $Functor(F)$. Thus each constructor of a type gives rise to a type which can be used to type both application of constructors and pattern-matches.

We can only associate new names to types when going under lambda abstractions or encountering let-bindings or case-expressions. Type-variables can only be associated with a kind when going under a quantified type. The remaining forms are not added or removed to the context during typing. Usually, these forms are elaborated from declarations given by the programmer in a pre-processing step before typing takes place.

5.2.2 Kinds of types

Types in CloFRP have kinds, and these kinds can be derived inductively, as formalized in Figure 21. Asking whether the type of a term is well-formed amounts to asking whether it correctly derives a kind, and if the kind is what we expected, which depends on the context (usually we expect $\star$). Note that the KIND$\to$ rule disallows functions on clock-variables as specified in [4]. The KINDTUPLE rule handles n-ary tuples in the obvious way. Otherwise, the rules are pretty standard kind-inference rules.

$\boxed{\Psi \vdash A \Rightarrow \chi}$ Under context Ψ, type A has kind χ

$$\frac{(\alpha : \chi) \in \Psi}{\Psi \vdash \alpha \Rightarrow \chi}\ \textsc{KindVar} \qquad \frac{(\mathcal{T} : \chi) \in \Psi}{\Psi \vdash \mathcal{T} \Rightarrow \chi}\ \textsc{KindFree} \qquad \frac{\Psi \vdash A \Rightarrow \star \qquad \Psi \vdash B \Rightarrow \star}{\Psi \vdash A \to B \Rightarrow \star}\ \textsc{Kind}\to$$

$$\frac{\Psi \vdash A_1 \Rightarrow \star \qquad \ldots \qquad \Psi \vdash A_n \Rightarrow \star}{\Psi \vdash \langle A_1, \ldots, A_n \rangle \Rightarrow \star}\ \textsc{KindTuple} \qquad \frac{\Psi, \alpha : \chi \vdash A \Rightarrow \star}{\Psi \vdash \forall \alpha : \chi.\, A \Rightarrow \star}\ \textsc{Kind}\forall$$

$$\frac{\Psi \vdash F \Rightarrow (\chi_1 \to \chi_2) \qquad \Psi \vdash B \Rightarrow \chi_1}{\Psi \vdash F\, B \Rightarrow \chi_2}\ \textsc{KindApp} \qquad \frac{\Psi \vdash F \Rightarrow (\star \to \star)}{\Psi \vdash \mathsf{Fix}\, F \Rightarrow \star}\ \textsc{KindFix}$$

$$\frac{\Psi \vdash \kappa \Rightarrow \mathsf{c} \qquad \Psi \vdash A \Rightarrow \star}{\Psi \vdash \triangleright^\kappa A \Rightarrow \star}\ \textsc{Kind}\triangleright^\kappa$$

Figure 21: Deriving kinds of types in CloFRP.

5.2.3 Subtyping

The subtyping rules of CloFRP (Figure 22) are pretty much identical to the declarative subtyping rules of [30], except extended to work with the larger source language. A tuple is a subtype of another tuple iff all the components are subtypes of each other. Type-application, recursive types and delayed computations all simply proceed inductively. The rest are identical to the declarative subtyping rules in [30].

$\boxed{\Psi \vdash A \leq B}$ Under context Ψ, type A is a subtype of B

$$\frac{\alpha : \chi \in \Psi}{\Psi \vdash \alpha \leq \alpha} \leq\textsc{Var} \qquad \frac{\mathcal{T} : \chi \in \Psi}{\Psi \vdash \mathcal{T} \leq \mathcal{T}} \leq\textsc{Free} \qquad \frac{\Psi \vdash B_1 \leq A_1 \qquad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2} \leq\!\to$$

$$\frac{\Psi \vdash \tau \Mapsto \chi \qquad \Psi \vdash [\tau/\alpha]A \leq B}{\Psi \vdash \forall \alpha : \chi.\, A \leq B} \leq\forall\mathrm{L} \qquad \frac{\Psi, \beta : \chi \vdash A \leq B}{\Psi \vdash A \leq \forall \beta : \chi.\, B} \leq\forall\mathrm{R}$$

$$\frac{\Psi \vdash A_1 \leq B_1 \qquad \ldots \qquad \Psi \vdash A_n \leq B_n}{\Psi \vdash \langle A_1, \ldots, A_n \rangle \leq \langle B_1, \ldots, B_n \rangle} \leq\textsc{Tuple} \qquad \frac{\Psi \vdash F \leq G \qquad \Psi \vdash A \leq B}{\Psi \vdash F\, A \leq G\, B} \leq\textsc{App}$$

$$\frac{\Psi \vdash F \leq G}{\Psi \vdash \mathsf{Fix}\, F \leq \mathsf{Fix}\, G} \leq\textsc{Fix} \qquad \frac{\Psi \vdash \kappa_1 \leq \kappa_2 \qquad \Psi \vdash A \leq B}{\Psi \vdash \triangleright^{\kappa_1} A \leq \triangleright^{\kappa_2} B} \leq\!\triangleright^{\kappa}$$

Figure 22: Subtyping in CloFRP.

5.2.4 Declarative inference rules

The inference rules for CloFRP (Figure 23) again closely resemble those of [30]. Similarly, the rules are type-checking, type-synthesis and application-synthesis, along with a separate judgment for patterns and case expressions.

$\boxed{\Psi \vdash e \Leftarrow A}$ Under context Ψ, e checks against type A

$\boxed{\Psi \vdash e \Rightarrow A}$ Under context Ψ, e synthesizes type A

$\boxed{\Psi \vdash A \bullet e \Rrightarrow C}$ Under context Ψ, applying a function of type A to e synthesizes type C

$\boxed{\Psi \vdash (\rho : A \longrightarrow e) \Lleftarrow B}$ Under context Ψ, check branch $(\rho \longrightarrow e)$ with pattern-type A and body-type B

$\boxed{\Psi \vdash \rho \not\Leftarrow A \dashv \Psi'}$ Under context Ψ, check pattern ρ with type A yielding new context Ψ'

$$\frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A}\ \textsc{Var} \qquad \frac{(\mathcal{C} : A) \in \Psi}{\Psi \vdash \mathcal{C} \Rightarrow A}\ \textsc{Constr}\Rightarrow \qquad \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B \dashv \upsilon}{\Psi \vdash e \Leftarrow B}\ \textsc{Sub}$$

$$\frac{\Psi \vdash A \Mapsto \star \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A}\ \textsc{Anno} \qquad \frac{\Psi, \alpha : \chi \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall \alpha : \chi.\ A}\ \forall\text{I} \qquad \frac{\Psi, x :_\lambda A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\, e \Leftarrow A \to B}\ \to\text{I}$$

$$\frac{\Psi \vdash \tau \Mapsto \chi \quad \Psi \vdash [\tau/\alpha]A \bullet e \Rrightarrow C}{\Psi \vdash (\forall \alpha : \chi.\ A) \bullet e \Rrightarrow C}\ \forall\textsc{App} \qquad \frac{\Psi \vdash \sigma \to \tau \Mapsto \star \quad \Psi, x :_\lambda \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\, e \Rightarrow \sigma \to \tau}\ \to\text{I}\Rightarrow$$

$$\frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\, e_2 \Rightarrow C}\ \to\text{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash (A \to C) \bullet e \Rrightarrow C}\ \to\textsc{App}$$

$$\frac{\Psi \vdash e_1 \Rightarrow A_1 \ \ldots\ \Psi \vdash e_n \Rightarrow A_n}{\Psi \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle A_1, \ldots, A_n \rangle}\ \textsc{TupleI}\Rightarrow \qquad \frac{\Psi \vdash e_1 \Leftarrow A_1 \ \ldots\ \Psi \vdash e_n \Leftarrow A_n}{\Psi \vdash \langle e_1, \ldots, e_n \rangle \Leftarrow \langle A_1, \ldots, A_n \rangle}\ \textsc{TupleI}$$

$$\frac{\Psi \vdash \kappa \Mapsto \mathsf{c} \quad \Psi, x :_\lambda \kappa \vdash e \Rightarrow A}{\Psi \vdash \gamma(x : \kappa).\, e \Rightarrow \triangleright^\kappa A}\ \triangleright^\kappa\text{I}\Rightarrow \qquad \frac{\Psi \vdash \kappa \Mapsto \mathsf{c} \quad \Psi \vdash x \Leftarrow \kappa}{\Psi \vdash (\triangleright^\kappa A) \bullet [x] \Rrightarrow A}\ \triangleright^\kappa\textsc{App}$$

$$\frac{\Psi \vdash \kappa \Mapsto \mathsf{c} \quad \mathit{stable}_\kappa(\Psi)}{\Psi \vdash (\triangleright^\kappa A) \bullet [\diamond] \Rrightarrow A}\ \triangleright^\kappa\textsc{App}\diamond \qquad \frac{\Psi \vdash e \Rightarrow \forall (\alpha : \chi).\ B \quad A \Mapsto \chi}{\Psi \vdash e\,\{A\} \Rightarrow [A/\alpha]B}\ \textsc{TApp}$$

$$\frac{}{\Psi \vdash \mathsf{fold} \Rightarrow \forall (\alpha : \star \to \star).\ \alpha\,(\mathsf{Fix}\,\alpha) \to \mathsf{Fix}\,\alpha}\ \textsc{Fold}\Rightarrow \qquad \frac{}{\Psi \vdash \mathsf{unfold} \Rightarrow \forall (\alpha : \star \to \star).\ \mathsf{Fix}\,\alpha \to \alpha\,(\mathsf{Fix}\,\alpha)}\ \textsc{Unfold}\Rightarrow$$

$$\frac{}{\Psi \vdash \mathsf{fix} \Rightarrow \forall (\kappa : \mathsf{c}), \alpha.\ (\triangleright^\kappa \alpha \to \alpha) \to \alpha}\ \textsc{Fix} \qquad \frac{\mathit{Functor}(F) \in \Psi}{\Psi \vdash \mathsf{fmap}_F \Rightarrow \forall \alpha, \beta.\ (\alpha \to \beta) \to F\,\alpha \to F\,\beta}\ \textsc{Fmap}$$

$$\frac{\mathit{Functor}(F) \in \Psi}{\Psi \vdash \mathsf{primRec}_F \Rightarrow \forall \alpha.\ F\,\langle \mathsf{Fix}\,F, \alpha \rangle \to \mathsf{Fix}\,F \to \alpha}\ \textsc{PrimRec}$$

$$\frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash \tau \Mapsto \star \quad \Psi \vdash A \leq \tau \quad \Psi \vdash (\rho_1 : \tau \longrightarrow e_1) \Lleftarrow B \ \ldots\ \Psi \vdash (\rho_n : \tau \longrightarrow e_n) \Lleftarrow B}{\Psi \vdash \mathsf{case}\ e\ \mathsf{of}\ (\rho_1 \longrightarrow e_1, \ldots, \rho_n \longrightarrow e_n) \Leftarrow B}\ \textsc{Case}$$

$$\frac{\Psi \vdash \rho \not\Leftarrow A \dashv \Psi' \quad \Psi' \vdash e \Leftarrow B}{\Psi \vdash (\rho : A \longrightarrow e) \Lleftarrow B}\ \textsc{Branch} \qquad \frac{\Psi \vdash \rho_1 \not\Leftarrow A_1 \dashv \Psi_1 \ \ldots\ \Psi_{n-1} \vdash \rho_n \not\Leftarrow \dashv \Psi_n}{\Psi \vdash \langle \rho_1, \ldots, \rho_n \rangle \not\Leftarrow \langle A_1, \ldots, A_n \rangle \dashv \Psi_n}\ \textsc{PatTuple}$$

$$\frac{}{\Psi \vdash x \not\Leftarrow A \dashv \Psi, x :_= A}\ \textsc{PatBind}$$

$$\frac{(\mathcal{C} : A') \in \Psi \qquad \Psi \vdash \rho_1 \not\Leftarrow B_1 \dashv \Psi_1 \ \ldots\ \Psi_{n-1} \vdash \rho_n \not\Leftarrow B_n \dashv \Psi_n \qquad \Psi \vdash A' \leq B_1 \to \cdots \to B_n \to A}{\Psi \vdash (\mathcal{C}\ \rho_1, \ldots, \rho_n) \not\Leftarrow A \dashv \Psi_n}\ \textsc{PatConstr}$$

$$\frac{\Psi \vdash e' \Rightarrow A \quad \Psi, x :_= A \vdash e \Leftarrow B}{\Psi \vdash \mathsf{let}\ x\ = e'\ \mathsf{in}\ e \Leftarrow B}\ \textsc{Let} \qquad \frac{\Psi \vdash \mathsf{case}\ e'\ \mathsf{of}\ (\rho \longrightarrow e) \Leftarrow B \quad \rho \neq x}{\Psi \vdash \mathsf{let}\ \rho\ = e'\ \mathsf{in}\ e \Leftarrow B}\ \textsc{LetPat}$$

Figure 23: Inference rules for CloFRP.

Constr⇒ infers the type of constructors in a similar fashion to bound names. The introduction rule for delayed computations $\triangleright^\kappa$I is similar to the rule of lambda abstractions, but requires annotations on the tick-variable, and checks that the annotated clock is of the correct kind. There are two new application-synthesis rules for delayed computations: $\triangleright^\kappa$App forces a delayed computation via application to a tick-variable, while $\triangleright^\kappa\diamond$ applies a tick-constant to the computation. Here, the $stable_k(\Psi)$ premise serves the role of the κ-*stable* judgment in [4], which asserts that Ψ does not have κ free (i.e. there are no $x :_\lambda \kappa$ bindings in Ψ).

TApp models explicit type applications; that is, if you have a term of a polymorphic type, you can explicitly instantiate the outer-most quantified type-variable. This construct also models clock-application from CloTT, as clock-variables are just types of kind c in this sytem. Furthermore, explicit type-application also allows the programmer to express some forms of impredicative polymorphism. The programmer can choose to explicitly instantiate a type-variable with a polymorphic type if he so wishes. Since there is no inference involved, we do not run into the usual problems associated with inferring impredicative higher-rank types (which is undecidable [35]).

Fold and Unfold infers the types of fold and unfold respectively. Since we are formulating a system with inference, we do not have to use the usual application-form for these combinators, but can abstract over their types in the source language instead.

Fix is the guarded fixpoint of simply-typed CloTT. Again, we can formulate the rule without application by giving it a polymorphic type.

Fmap encodes mapping over a value of a functorial type. The type of F must be given explicitly here, but could as well have been inferred in the declarative system – however, when we formulate the algorithmic system, we shall not include a mechanism for deducing functoriality of inferred types, so we keep the annotation in the declarative system to make the gap between them slightly smaller. Also note that our system does not include a mechanism to abstract over functors. We cannot write, as you would be able to in Haskell, `Functor f => a -> f b -> f b`. This is not an inherent limitation of the inference system, nor the underlying calculus. It is simply out of scope for this project.

The only true purpose of `fmap` is to enable proper evaluation of the primitive recursion combinator, which is modeled by the rule PrimRec. The rule essentially assigns the $\mathsf{primRec}_F$ construct the exact type one would give it if implementing it in Haskell using general recursion, except we cannot abstract over functors, so the functorial type F must be given explicitly.

Case-expressions (the Case rule) are a bit complicated. A case expressions consists of a *scrutinee* e which is pattern-matched on, and a list of branches. Each branch is a pattern ρ_i and a body e_i. If the scrutinee e infers the possibly polymorphic type A, then each pattern ρ_i in the branches must check against that type, and extend the context with appropriate bindings, before checking the corresponding body e_i against the return type B (rule Branch). The complexity arises if A is a polymorphic type. Then, all its quantifiers must be instantiated with a guessed type before proceeding to check the patterns against it. We model this by guessing a monomorphic type τ of which A is a subtype – that is, τ is a specialization of A – and then checking against this type instead. Intuitively, there is no way to pattern match directly on a polymorphic definition, as it has no syntactic form. In SystemF it would be a type abstraction $\Lambda\alpha.\, e$, and would also need to be applied a type to eliminate the Λ before matching on e.

The $\not\Rightarrow$ judgment models pattern matching by checking a pattern against a type, and returning a new context with pattern-bound names. PatBind simply binds a type to a name, while PatTuple handles tuple-patterns by checking each sub-pattern in sequence. PatConstr is somewhat more complicated, and handles checking constructor-patterns. It first looks up the pattern in the context which carries information about the type of the pattern A'. We must then guess the correct types of all the sub-patterns $B_1, \ldots, B_n$, such that we can construct a type $B_1 \to \cdots \to B_n \to A$ that is a super-type (more specific than) A'.

For example, the process of checking the program in Listing 2 is roughly as follows:

- In a pre-processing step, the data-type declaration on line 1 gives a context Ψ with $\mathsf{Nothing} : (\forall a.\, \mathsf{Maybe}\, a) \in \Psi$ and $\mathsf{Just} : (\forall a.\, a \to \mathsf{Maybe}\, a) \in \Psi$.

- After going under the lambdas, the rule for case-expressions can be used. It will infer the type of `x` to be Maybe a. Since this type is already mono-morphic, we will have that $\tau \equiv A$. There are then two case-branches that are analyzed.
- The first case simply checks if the type of the constructor Nothing is a subtype of the type of `x`, giving the judgment $(\forall a.\, \mathsf{Maybe}\, a) \leq \mathsf{Maybe}\, a$ which is trivially true. Then it checks that `def` checks against `a` which it naturally does by one application of the VAR rule.
- The second case will have to guess the type of `x'` (call it B) such that the type of Just (which is $\forall a.a \to \mathsf{Maybe}\, a$) is a subtype of $B \to \mathsf{Maybe}\, a$. The obvious guess here is to set $B = a$. That means we will check $x' \Leftarrow a$ in a context where $x' : a$ and the typing checks out by one application of VAR.

```
1  data Maybe a = Nothing | Just a.
2
3  orElse : forall a. Maybe a -> a -> a.
4  orElse = \x def ->
5    case x of
6    | Nothing -> def
7    | Just x' -> x'.
```

Listing 2: An example program with pattern matching.

Finally, let-bindings are encoded with two rules. LET handles the simple case where the pattern ρ is just a name x, and we simply bind the type of e' to x. LETPAT handles the case where the let-binding does a form of pattern matching. In that case, the rule simply desugars the let-binding to a case expression with one case. The reason we have two rules is that it is possible to bind names to polymorphic types with let-expressions, but *not* with case-expressions. Thus, if $\Psi \vdash e' \Rightarrow \forall \alpha.A$ then $\Psi \vdash \mathsf{let}\, x = e'\, \mathsf{in}\, e \Leftarrow B$ continues with $\Psi, (x :_= \forall \alpha.\ A) \vdash e \Leftarrow B$. In contrast, $\Psi \vdash \mathsf{case}\ e'\ \mathsf{of}\ (x \longrightarrow e) \Leftarrow B$ fully applies $\forall \alpha.\ A$ before binding it, eventually leading to $\Psi, (x :_= [\tau/\alpha]A) \vdash e \Leftarrow B$.

This concludes the section on the declarative specification of CloFRP. While the declarative rules describe the language adequately, they do not directly yield an algorithm for checking programs written in CloFRP, because they enjoy guessing far too much.

5.3 Algorithmic inference for CloFRP

To describe an algorithm for type-checking CloFRP programs, we shall employ the same method as [30], where we introduce existential type variables and ordered contexts to "defer" the choice of type-instantiations to a later time. Once again, the algorithmic system is quite a bit more verbose than that of [30] as it deals with a substantially larger and more complicated language, but is otherwise more or less a straightforward extension of their algorithmic system, much in the same way that our declarative system was an extension of theirs.

5.3.1 Syntax

The syntax of the algorithmic system is nearly identical to the declarative system. Types are the same, but now also contain the existential type-variable $\hat{\alpha}$. Unlike in the declarative specification, we shall choose to split the context in two: a *global* and a *local* context. The global context Φ contains assertions that do not change throughout typing, such as constructors and patterns. The Φ context is not especially interesting, and so we shall omit it from the judgment and use it as an implicit object. There is one change in the global context compared to the subset of the declarative context that it describes, namely that patterns are no longer associated to some type, but to a list of bound existentials $\vec{\hat{\alpha}}$ argument-types $\hat{B}_i$ and a result type $\hat{\tau}$. This representation will make pattern matching specification slightly more elegant. The local algorithmic context is now ordered, and can contain existential variables $\hat{\alpha} : \chi$, monotype

solutions to existentials $\hat{\alpha} : \chi = \tau$, and markers $\blacktriangleright_{\hat{\alpha}}$. Bound names are still distinguished between their binders; lambda-bound names are written $x :_\lambda A$ while let- and pattern-bound variables are $x :_= A$. The grammar for algorithmic contexts Γ and Φ can be seen in Figure 24.

$$
\begin{array}{llll}
\text{Global contexts} & \Phi & ::= & \cdot \mid \Gamma, \mathcal{C} : A \mid \Gamma, \mathcal{T} : \chi \mid \Gamma, \mathcal{P} : \forall \vec{\hat{\alpha}}.\ \hat{B}_1 \to \cdots \to \hat{B}_n \to \hat{\tau} \mid Functor(F) \\
\text{Ordered contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, x :_\lambda A \mid \Gamma, x :_= A \mid \Gamma, \alpha : \chi \mid \Gamma, \hat{\alpha} : \chi \mid \Gamma, \hat{\alpha} : \chi = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}}
\end{array}
$$

Figure 24: Syntax for algorithmic contexts.

Precisely as in [30], algorithmic contexts are only well-formed if every context-element only depends on type-variables that are located "before" itself. This notion is formalized in Figure 25. Note that we write simply $x : A$ when we do not care if x was bound by a lambda or a pattern.

$\boxed{\Gamma\ ctx}$ Algorithmic context Γ is well-formed

$$\frac{}{\cdot\ ctx}\ \textsc{EmptyCtx} \qquad \frac{\Gamma\ ctx \qquad \alpha \notin \mathsf{dom}\,(\Gamma)}{\Gamma, \alpha : \chi\ ctx}\ \textsc{UVarCtx}$$

$$\frac{\Gamma\ ctx \qquad x \notin \mathsf{dom}\,(\Gamma) \qquad \Gamma \vdash A \Rightarrow \star}{\Gamma, x : A\ ctx}\ \textsc{VarCtx} \qquad \frac{\Gamma\ ctx \qquad \hat{\alpha} \notin \mathsf{dom}\,(\Gamma)}{\Gamma, \hat{\alpha} : \chi\ ctx}\ \textsc{EVarCtx}$$

$$\frac{\Gamma\ ctx \qquad \hat{\alpha} \notin \mathsf{dom}\,(\Gamma) \qquad \Gamma \vdash \tau \Rightarrow \chi}{\Gamma, \hat{\alpha} : \chi = \tau\ ctx}\ \textsc{SolvedEVarCtx}$$

$$\frac{\Gamma\ ctx \qquad \blacktriangleright_{\hat{\alpha}} \notin \Gamma \qquad \hat{\alpha} \notin \mathsf{dom}\,(\Gamma)}{\Gamma, \blacktriangleright_{\hat{\alpha}}\ ctx}\ \textsc{MarkerCtx}$$

Figure 25: Well-formedness for algorithmic contexts.

5.3.2 Kinds of types

Types also have kinds, in exactly the same way as the declarative system, except for two new rules that handle existential type variables. The kind inference judgment can be seen in Figure 26.

$\boxed{\Gamma \vdash A \Rightarrow \chi}$ Under context Γ, type A has kind χ

$$\frac{(\alpha : \chi) \in \Gamma}{\Gamma \vdash \alpha \Rightarrow \chi}\ \textsc{KindVar} \qquad \frac{(\mathcal{T} : \chi) \in \Phi}{\Gamma \vdash \mathcal{T} \Rightarrow \chi}\ \textsc{KindFree} \qquad \frac{\Gamma \vdash A \Rightarrow \star \qquad \Gamma \vdash B \Rightarrow \star}{\Gamma \vdash A \to B \Rightarrow \star}\ \textsc{Kind}{\to}$$

$$\frac{\Gamma \vdash A_1 \Rightarrow \star \qquad \ldots \qquad \Gamma \vdash A_n \Rightarrow \star}{\Gamma \vdash \langle A_1, \ldots, A_n \rangle \Rightarrow \star}\ \textsc{KindTuple} \qquad \frac{\Gamma, \alpha : \chi \vdash A \Rightarrow \star}{\Gamma \vdash \forall \alpha : \chi.\, A \Rightarrow \star}\ \textsc{Kind}\forall$$

$$\frac{\Gamma \vdash F \Rightarrow (\chi_1 \to \chi_2) \qquad \Gamma \vdash B \Rightarrow \chi_1}{\Gamma \vdash F\ B \Rightarrow \chi_2}\ \textsc{KindApp} \qquad \frac{\Gamma \vdash F \Rightarrow (\star \to \star)}{\Gamma \vdash \mathsf{Fix}\ F \Rightarrow \star}\ \textsc{KindFix}$$

$$\frac{\Gamma \vdash \kappa \Rightarrow \mathsf{c} \qquad \Gamma \vdash A \Rightarrow \star}{\Gamma \vdash \triangleright^\kappa A \Rightarrow \star}\ \textsc{Kind}{\triangleright^\kappa} \qquad \frac{\hat{\alpha} : \chi \in \Gamma}{\Gamma \vdash \hat{\alpha} \Rightarrow \chi}\ \textsc{KindEVar} \qquad \frac{\hat{\alpha} : \chi = \tau \in \Gamma}{\Gamma \vdash \hat{\alpha} \Rightarrow \chi}\ \textsc{KindEVarSolved}$$

Figure 26: Kind inference in the algorithmic specification of CloFRP.

5.3.3 Subtyping

Again, the subtyping relation (Figure 27) is a mostly straightforward extension of the declarative subtyping rules, using the same schema as in [30]. Every guess of a type is replaced by introducing an existential type-variable in its stead.

$\boxed{\Gamma \vdash A <: B \dashv \Delta}$ Under context Γ, type A is a subtype of B, yielding output context Δ

$$\frac{}{\Gamma[\alpha:\chi] \vdash \alpha <: \alpha \dashv \Gamma[\alpha:\chi]}\ {<:}\textsc{Var} \qquad \frac{\mathcal{T} \in \Phi}{\Gamma \vdash \mathcal{T} <: \mathcal{T} \dashv \Gamma}\ {<:}\textsc{Free}$$

$$\frac{}{\Gamma[\hat{\alpha}:\chi] \vdash \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}:\chi]}\ {<:}\textsc{Exvar} \qquad \frac{\Gamma \vdash B_1 <: A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Delta}\ {<:}\to$$

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha}:\chi \vdash [\hat{\alpha}/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash \forall \alpha:\chi.\, A <: B \dashv \Delta}\ {<:}\forall\text{L} \qquad \frac{\Gamma, \alpha:\chi \vdash A <: B \dashv \Delta, \alpha:\chi, \Theta}{\Gamma \vdash A <: \forall \alpha:\chi.\, B \dashv \Delta}\ {<:}\forall\text{R}$$

$$\frac{\Gamma \vdash A_1 <: B_1 \dashv \Delta_1 \quad \dots \quad \Delta_{n-1} \vdash [\Delta_{n-1}]A_n <: [\Delta_{n-1}]B_n \dashv \Delta_n}{\Gamma \vdash \langle A_1, \dots, A_n \rangle <: \langle B_1, \dots, B_n \rangle \dashv \Delta_n}\ {<:}\textsc{Tuple}$$

$$\frac{\Gamma \vdash F <: G \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash F\ A <: G\ B \dashv \Delta}\ {<:}\textsc{App} \qquad \frac{\Gamma \vdash F <: G \dashv \Delta}{\Gamma \vdash \mathsf{Fix}\, F <: \mathsf{Fix}\, G \dashv \Delta}\ {<:}\textsc{Fix}$$

$$\frac{\Gamma \vdash \kappa_1 <: \kappa_2 \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash \triangleright^{\kappa_1} A <: \triangleright^{\kappa_2} B \dashv \Delta}\ {<:}\triangleright^{\kappa}$$

$$\frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}:\chi] \vdash \hat{\alpha}:\chi :\leqq A \dashv \Delta}{\Gamma[\hat{\alpha}:\chi] \vdash \hat{\alpha} <: A \dashv \Delta}\ {<:}\textsc{InstantiateL}$$

$$\frac{\hat{\alpha} \notin FV(A) \qquad \Gamma[\hat{\alpha}:\chi] \vdash A \leqq: \hat{\alpha}:\chi \dashv \Delta}{\Gamma[\hat{\alpha}:\chi] \vdash A <: \hat{\alpha} \dashv \Delta}\ {<:}\textsc{InstantiateR}$$

Figure 27: Algorithmic subtyping in CloFRP.

As in [30] we make sure to always apply intermediate contexts in the premises, making sure that types are always "most-solved". Components in tuple types are (arbitrarily) compared left-to-right. Finally, InstantiateL and InstantiateR switch to the instantiation judgments when an existential type-variable is compared with a non-existential type.

5.3.4 Instantiation

The instantiation rules (Figure 28) are mostly similar to the instantiation rules found in [30], but extended to work with the larger source language. In particular, instantiation also uses information about the kind of an existential to make sure that the kinds always match up. The left-instantiation rule $\Gamma \vdash \hat{\alpha}:\chi :\leqq A \dashv \Delta$ instantiates an existential variable $\hat{\alpha}$ of kind χ to a subtype (that is, a more-polymorphic type) of A. Dually, the right-instantiation rule $\Gamma \vdash A \leqq: \hat{\alpha}:\chi \dashv \Delta$ instantiates the existential to a supertype of A. The rules for solving an existential, and instantiating an existential to another existential, are identical to the ones found in [30], but with a few modifications to accomodate kinds. The same applies to the rule for instantiation to an arrow-type; the contravariant twist is still there, so instantiating the first refined existential switches to the dual instantiation judgment.

$\boxed{\Gamma \vdash \hat{\alpha} : \chi :\leqq A \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $\hat{\alpha} <: A$, with output context Δ

$$\frac{\Gamma \vdash \tau \Mapsto \chi}{\Gamma[\hat{\alpha} : \chi] \vdash \hat{\alpha} : \chi :\leqq \tau \dashv \Gamma[\hat{\alpha} : \chi = \tau]}\ \textsc{InstLSolve} \qquad \frac{\Gamma[\hat{\alpha} : \chi], \beta : \chi \vdash \hat{\alpha} : \chi :\leqq B \dashv \Delta, \beta : \chi, \Delta'}{\Gamma[\hat{\alpha} : \chi] \vdash \hat{\alpha} : \chi :\leqq \forall \beta : \chi.\ B \dashv \Delta}\ \textsc{InstLAllR}$$

$$\frac{}{\Gamma[\hat{\alpha} : \chi][\hat{\beta} : \chi] \vdash \hat{\alpha} : \chi :\leqq \hat{\beta} \dashv \Gamma[\hat{\alpha} : \chi][\hat{\beta} : \chi = \hat{\alpha}]}\ \textsc{InstLReach}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \star,\ \hat{\alpha}_1 : \star,\ \hat{\alpha} : \chi = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash A_1 \leqq: \hat{\alpha}_1 : \star \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 : \star :\leqq [\Theta] A_2 \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} : \star :\leqq A_1 \to A_2 \dashv \Delta}\ \textsc{InstLArr}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \Mapsto (\chi_1 \to \chi_2) \\ \Gamma[\hat{\alpha}_2 : \chi_1,\ \hat{\alpha}_1 : \chi_1 \to \chi_2,\ \hat{\alpha} : \chi_2 = \hat{\alpha}_1\ \hat{\alpha}_2] \vdash \hat{\alpha}_1 : \chi_1 \to \chi_2 :\leqq A_1 \dashv \Theta \end{array} \qquad \Theta \vdash \hat{\alpha}_2 : \chi_1 :\leqq [\Theta] A_2 \dashv \Delta}{\Gamma[\hat{\alpha} : \chi_2] \vdash \hat{\alpha} : \chi_2 :\leqq A_1\ A_2 \dashv \Delta}\ \textsc{InstLApp}$$

$$\frac{\Gamma[\hat{\alpha}_n : \star, \dots, \hat{\alpha}_1 : \star, \hat{\alpha} : \star = \langle \hat{\alpha}_1, \dots, \hat{\alpha}_n \rangle] \vdash \hat{\alpha}_1 : \star :\leqq A_1 \dashv \Delta_1\ \dots\ \Delta_{n-1} \vdash \hat{\alpha}_n : \star :\leqq [\Delta_{n-1}] A_n \dashv \Delta_n}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} : \star :\leqq \langle A_1, \dots, A_n \rangle \dashv \Delta_n}\ \textsc{InstLTuple}$$

$$\frac{\Gamma[\hat{\alpha}_1 : \star \to \star,\ \hat{\alpha} : \star = \mathsf{Fix}\, \hat{\alpha}_1] \vdash \hat{\alpha}_1 : \star \to \star :\leqq A \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} : \star :\leqq \mathsf{Fix}\, A \dashv \Delta}\ \textsc{InstLFix}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \star,\ \hat{\alpha}_1 : \mathsf{c},\ \hat{\alpha} = \triangleright^{\hat{\alpha}_1} \hat{\alpha}_2] \vdash \hat{\alpha}_1 : \mathsf{c} :\leqq \kappa \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 : \star :\leqq [\Theta] A \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} : \star :\leqq \triangleright^{\kappa} A \dashv \Delta}\ \textsc{InstL}\triangleright^{\kappa}$$

$\boxed{\Gamma \vdash A \leqq: \hat{\alpha} : \chi \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $A <: \hat{\alpha}$, with output context Δ

$$\frac{\Gamma[\hat{\alpha} : \chi] \vdash \tau \Mapsto \chi}{\Gamma[\hat{\alpha} : \chi] \vdash \tau \leqq: \hat{\alpha} : \chi \dashv \Gamma[\hat{\alpha} : \chi]}\ \textsc{InstRSolve} \qquad \frac{\Gamma[\hat{\alpha} : \chi], \blacktriangleright_{\hat{\beta}}, \hat{\beta} : \chi \vdash [\hat{\beta}/\beta] B \leqq: \hat{\alpha} : \star \dashv \Delta, \blacktriangleright_{\hat{\beta}}, \Delta'}{\Gamma[\hat{\alpha} : \chi] \vdash \forall \beta : \chi.\ B \leqq: \hat{\alpha} : \star \dashv \Delta}\ \textsc{InstRAllL}$$

$$\frac{}{\Gamma[\hat{\alpha} : \chi][\hat{\beta} : \chi] \vdash \hat{\beta} \leqq: \hat{\alpha} \dashv \Gamma[\hat{\alpha} : \chi][\hat{\beta} : \chi = \hat{\alpha}]}\ \textsc{InstRReach}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \star,\ \hat{\alpha}_1 : \star,\ \hat{\alpha} : \star = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \hat{\alpha}_1 : \star :\leqq A_1 \dashv \Theta \qquad \Theta \vdash [\Theta] A_2 \leqq: \hat{\alpha}_2 : \star \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash A_1 \to A_2 \leqq: \hat{\alpha} : \star \dashv \Delta}\ \textsc{InstRArr}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \Mapsto (\chi_1 \to \chi_2) \\ \Gamma[\hat{\alpha}_2 : \chi_1,\ \hat{\alpha}_1 : \chi_1 \to \chi_2,\ \hat{\alpha} : \chi_2 = \hat{\alpha}_1\ \hat{\alpha}_2] \vdash A_1 \leqq: \hat{\alpha}_1 : \chi_1 \to \chi_2 \dashv \Theta \end{array} \qquad \Theta \vdash [\Theta] A_2 :\leqq \hat{\alpha}_2 : \chi_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \chi_2] \vdash A_1\ A_2 \leqq: \hat{\alpha} : \chi_2 \dashv \Delta}\ \textsc{InstRApp}$$

$$\frac{\Gamma[\hat{\alpha}_n : \star, \dots, \hat{\alpha}_1 : \star, \hat{\alpha} : \star = \langle \hat{\alpha}_1, \dots, \hat{\alpha}_n \rangle] \vdash A_1 \leqq: \hat{\alpha}_1 : \star \dashv \Delta_1\ \dots\ \Delta_{n-1} \vdash [\Delta_{n-1}] A_n \leqq: \hat{\alpha}_n : \star \dashv \Delta_n}{\Gamma[\hat{\alpha} : \star] \vdash \langle A_1, \dots, A_n \rangle \leqq: \hat{\alpha} : \star \dashv \Delta_n}\ \textsc{InstRTuple}$$

$$\frac{\Gamma[\hat{\alpha}_1 : \star \to \star,\ \hat{\alpha} : \star = \mathsf{Fix}\, \hat{\alpha}_1] \vdash A \leqq: \hat{\alpha}_1 : \star \to \star \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \mathsf{Fix}\, A \leqq: \hat{\alpha} : \star \dashv \Delta}\ \textsc{InstRFix}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \star,\ \hat{\alpha}_1 : \mathsf{c},\ \hat{\alpha} = \triangleright^{\hat{\alpha}_1} \hat{\alpha}_2] \vdash \kappa \leqq: \hat{\alpha}_1 : \mathsf{c} \dashv \Theta \qquad \Theta \vdash [\Theta] A \leqq: \hat{\alpha}_2 : \star \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \triangleright^{\kappa} A \leqq: \hat{\alpha} : \star \dashv \Delta}\ \textsc{InstR}\triangleright^{\kappa}$$

Figure 28: Instantiation of existentials in CloFRP.

The new rules for type-application, tuples, recursive types and later-types follow the same pattern as the arrow-instantiation; the existential is refined and the rule then proceeds structurally on the type. Note that all the new type-formers are simply covariant, so they do not switch instantiation judgments in their premises. When an existential is articulated, we insert the type variable that we first proceed inductively on to the right of the other fresh existentials. For example, in the InstLArr rule, $\hat{\alpha}_1 : \star$ is to the right of $\hat{\alpha}_2 : \star$ in the premise. This does not really effect the result, but means that if the type-variables solve to each other, we should not hit the "Reaching" rules in the leafs, which is just slightly more natural.

5.3.5 Algorithmic inference rules

Finally, we present the algorithmic inference rules in Figures 30 and 31. Again, they are mostly a mechanical result of extending the algorithmic inference rules in [30] to work with our language, in the same way we extended the declarative rules. We leave out the inference rules for the primitives $\mathsf{fold}, \mathsf{unfold}, \mathsf{fmap}_F, \mathsf{primRec}_F$ for space reasons in this presentation, as they are basically identical to their declarative specification. Since we have two kinds of applications (normal application and tick-application), we also have two eliminating rules $\to$App and $\triangleright^\kappa$App, symmetrically to the declarative rules. However, we add an extra rule for tick-application to existential variables $\triangleright^\kappa\hat{\alpha}$App. Like in the normal existential application rule $\hat{\alpha}$App, we can articulate the shape of an existential type-variable if its associated term is applied with a tick-variable; then it must have a later-type form.

Pattern matching is somewhat more interesting. Case expressions must still fully instantiate the type of the scrutinee, but now we cannot just guess anymore. Instead, we "introduce" all quantified variables in A with fresh existentials using *intros*. Therefore, *intros* returns both the instantiated type and an output context with the fresh existentials appended.

Branches (rule Branch) introduce a marker $\blacktriangleright_\beta$. The name of the marker is not important, as long as it is not already in the context. The marker only serves to discard the bindings that the pattern match brings into scope when we exit the branch. Note that branches can still reveal new information about existentials that are already in scope. Since these existentials will be located to the left of the marker, we do not lose this information.

Matching on constructors (rule PatConstr) follows the same general structure as its declarative counterpart, but is slightly more involved as we can no longer guess types for the sub-patterns. Instead, we use the information in the global context Φ to deduce the types for the sub-patterns. Importantly, matching on a value of an applied type-constructor should reveal information about the concrete type it is applied to. For example, consider the `Maybe` type from Haskell, defined in CloFRP as `data Maybe a = Nothing | Just a`.

Such a declaration gives rise to two patterns in Φ namely $\mathsf{Nothing} : \forall\hat{\alpha} : \star.\ \mathsf{Maybe}\,\hat{\alpha}$ and $\mathsf{Just} : \forall\hat{\alpha} : \star.\ \hat{\alpha} \to \mathsf{Maybe}\,\hat{\alpha}$. As such, if $e \Rightarrow \mathsf{Maybe}\,\mathsf{1}$ then matching on e as `(case e of | Just x -> x) : 1` would result in the derivation in Figure 29.

$$\text{Let } \Delta \text{ denote } \Gamma, \blacktriangleright_\beta, \hat{\alpha} : \chi = 1 \quad \text{and} \quad \Delta' \text{ denote } \Delta, x :_= 1$$

$$\frac{\Gamma \vdash e \Rightarrow \mathsf{Maybe}\,\mathsf{1} \qquad \dfrac{T_1 \qquad \Delta' \vdash x \Leftarrow \dashv \Delta'}{\Gamma \vdash (\mathsf{Just}\,x : \mathsf{Maybe}\,\mathsf{1} \longrightarrow x) \twoheadleftarrow 1 \vdash \Gamma}\ \textsc{Branch}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{Just}\,x \longrightarrow x) \Leftarrow 1 \dashv \Gamma}\ \textsc{Case}$$

$$T_1 = \frac{(\mathsf{Just} : \forall\hat{\alpha} : \chi.\ \hat{\alpha} \to \mathsf{Maybe}\,\hat{\alpha}) \in \Phi \qquad \Gamma, \blacktriangleright_\beta, \hat{\alpha} : \chi \vdash \mathsf{Maybe}\,\hat{\alpha} <: \mathsf{Maybe}\,\mathsf{1} \dashv \Delta \qquad \Delta \vdash x \not\Leftarrow 1 \dashv \Delta'}{\Gamma, \blacktriangleright_\beta \vdash \mathsf{Just}\,x \not\Rightarrow \mathsf{Maybe}\,\mathsf{1} \dashv \Delta'}$$

Figure 29: Example derivation with pattern matching.

Finally, let-bindings are modeled by two rules in the same spirit as in the declarative formulation, namely Let and LetPat. If the pattern is just a name, we simply bind that name to the type of e'. On the other hand, if the let-binding pattern matches on e', we simply type it as the equivalent case-expression.

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \textsc{Var} \qquad \frac{(\mathcal{C} : A) \in \Phi}{\Gamma \vdash \mathcal{C} \Rightarrow A \vdash \Gamma}\ \textsc{Constr}\Rightarrow \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow A \dashv \Delta}\ \textsc{Anno}$$

$$\frac{\Gamma, \alpha : \chi \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall \alpha : \chi.\ A \dashv \Delta}\ \forall\mathrm{I} \qquad \frac{\Gamma, \hat{\alpha} : \chi \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha : \chi.\ A \bullet e \Rrightarrow C \dashv \Delta}\ \forall\textsc{App}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ \to\mathrm{I} \qquad \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \textsc{Sub}$$

$$\frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta}\ \to\mathrm{I}\Rightarrow \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow C \dashv \Delta}\ \to\mathrm{E}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \star, \hat{\alpha}_1 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2 \dashv \Delta}\ \hat{\alpha}\textsc{App}$$

$$\frac{\Gamma[\hat{\alpha}_2 : \chi, \hat{\alpha}_1 : \mathsf{c}, \hat{\alpha} : \star = \triangleright^{\hat{\alpha}_1} \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \bullet [x] \Rrightarrow \hat{\alpha}_2 \dashv \Delta}\ \triangleright^{\kappa}\hat{\alpha}\textsc{App} \qquad \frac{\Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Rrightarrow C \dashv \Delta}\ \to\textsc{App}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \dashv \Delta_1 \ \ldots\ \Delta_{n-1} \vdash e_n \Rightarrow A_n \dashv \Delta_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle \Rightarrow \langle A_1, \ldots, A_n \rangle \dashv \Delta_n}\ \textsc{TupleI}\Rightarrow$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A_1 \dashv \Delta_1 \ \ldots\ \Delta_{n-1} \vdash e_n \Leftarrow A_n \dashv \Delta_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle \Leftarrow \langle A_1, \ldots, A_n \rangle \dashv \Delta_n}\ \textsc{TupleI}$$

$$\frac{\Gamma \vdash \kappa \Mapsto \mathsf{c} \qquad \Gamma, x :_{\lambda} \kappa \vdash e \Rightarrow A \dashv \Delta}{\Gamma \vdash \gamma(x : \kappa).\, e \Rightarrow \triangleright^{\kappa} A \dashv \Delta}\ \triangleright^{\kappa}\mathrm{I}\Rightarrow \qquad \frac{\Gamma \vdash e \Rightarrow \forall(\alpha : \chi).\ B \dashv \Delta \qquad \Delta \vdash A \Mapsto \chi}{\Gamma \vdash e\,\{A\} \Rightarrow [A/\alpha]B \dashv \Delta}\ \textsc{TApp}$$

$$\frac{\Gamma \vdash \kappa \Mapsto \mathsf{c} \qquad \Gamma \vdash x \Leftarrow \kappa \dashv \Delta}{\Gamma \vdash (\triangleright^{\kappa} A) \bullet [x] \Rrightarrow A \dashv \Delta}\ \triangleright^{\kappa}\textsc{App} \qquad \frac{\Gamma \vdash \kappa \Mapsto \mathsf{c} \qquad \mathit{stable}_{\kappa}(\Gamma)}{\Gamma \vdash (\triangleright^{\kappa} A) \bullet [\diamond] \Rrightarrow A \dashv \Gamma}\ \triangleright^{\kappa}\textsc{App}\diamond$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow A \qquad (A', \Theta) = \mathit{intros}(A, \Gamma) \\ \Theta \vdash (\rho_1 : [\Theta]A' \longrightarrow e_1) \lll [\Theta]B \vdash \Delta_1 \ \ldots\ \Delta_{n-1} \vdash (\rho_n : [\Delta_{n-1}]A' \longrightarrow e_n) \lll B \dashv \Delta_n\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ (\rho_1 \longrightarrow e_1, \ldots, \rho_n \longrightarrow e_n) \Leftarrow B \dashv \Delta_n}\ \textsc{Case}$$

Figure 30: Algorithmic inference rules of CloFRP (part 1).

$\boxed{\Gamma \vdash (\rho : A \longrightarrow e) \mathrel{-\!\!\ll} B \dashv \Delta}$ Under context Γ, check branch $(\rho \longrightarrow e)$ with pattern-type A and body-type B, with output context Δ

$\boxed{\Gamma \vdash \rho \not\Leftarrow A \dashv \Delta}$ Under context Γ, check pattern ρ with type A yielding new context Δ

$$\frac{\Gamma, \blacktriangleright_\beta \vdash \rho \not\Leftarrow A \dashv \Theta \qquad \Theta \vdash e \Leftarrow B \dashv \Delta, \blacktriangleright_\beta, \Delta'}{\Gamma \vdash (\rho : A \longrightarrow e) \mathrel{-\!\!\ll} B \dashv \Delta}\ \textsc{Branch} \qquad \frac{}{\Gamma \vdash x \not\Leftarrow A \dashv \Gamma, x :_= A}\ \textsc{PatBind}$$

$$\frac{\Gamma \vdash \rho_1 \not\Leftarrow A_1 \dashv \Delta_1 \ \ldots \ \Delta_{n-1} \vdash \rho_n \not\Leftarrow \dashv \Delta_n}{\Gamma \vdash \langle \rho_1, \ldots, \rho_n \rangle \not\Leftarrow \langle A_1, \ldots, A_n \rangle \dashv \Delta_n}\ \textsc{PatTuple}$$

$$\frac{\begin{array}{c}(\mathcal{P} : \forall \vec{\hat{\alpha}}.\ \hat{B}_1 \to \cdots \to \hat{B}_n \to \hat{\tau}) \in \phi \\ \Gamma, \vec{\hat{\alpha}} \vdash \hat{\tau} <: A \dashv \Delta \qquad \Delta \vdash \rho_1 \not\Leftarrow [\Delta]\hat{B}_1 \dashv \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash \rho_n \not\Leftarrow [\Delta_{n-1}]\hat{B}_n \dashv \Delta_n\end{array}}{\Gamma \vdash \mathcal{P}\ \rho_1, \ldots, \rho_n \not\Leftarrow A \dashv \Delta_n}\ \textsc{PatConstr}$$

$$\frac{\Gamma \vdash e' \Rightarrow A \dashv \Theta \qquad \Theta, x :_= [\Theta]A \vdash e \Leftarrow [\Theta]B \dashv \Delta}{\Gamma \vdash \mathsf{let}\ x\ = e'\ \mathsf{in}\ e \Leftarrow B \dashv \Delta}\ \textsc{Let}$$

$$\frac{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ (\rho \longrightarrow e) \Leftarrow B \vdash \Gamma \qquad \rho \neq x}{\Gamma \vdash \mathsf{let}\ \rho\ = e'\ \mathsf{in}\ e \Leftarrow B \dashv \Delta}\ \textsc{LetPat}$$

Figure 31: Algorithmic inference rules of CloFRP (part 2).

This concludes the specification of the type-inference algorithm used by CloFRP. We shall now show some examples of CloFRP programs.

5.4 CloFRP examples

First, we shall declare some data-types that many of the examples will use.

```
1 data StreamF (k : Clock) a f = Cons a (|>k f) deriving Functor.
2 type Stream (k : Clock) a = Fix (StreamF k a).
3 data CoStream a = Cos (forall (kappa : Clock). Stream kappa a).
4 data ListF a f = Nil | LCons a f deriving Functor.
5 type List a = Fix (ListF a).
6 data Unit = MkUnit.
7 data Bool = True | False.
```

Here we declare the guarded streams over clocks by first writing its "base-functor" `StreamF` that factors out its recursive nature. We then create a type-synonym `Stream` to express its fixpoint solution. This is the only way to construct recursive data-types in CloFRP. We model coinductive streams by creating a new data-type `CoStream` that wraps the forall-quantification. This is a classic trick from Haskell to control when type-parameters are instantiated during type-checking. We model lists in a similar manner to Streams. Notice that `ListF` does not guard its functorial type variable `f` in a $\triangleright^\kappa$ type. This means that any values of `List a` will be finite and well-founded, and we can only use primitive recursion to recurse over such values. Finally, some non-interesting standard types are declared.

5.4.1 Productivity in the type-system

We can encode the `map` and `maap` functions from [3] in CloFRP (Listing 3).

```
-- in Haskell:
-- map f (x : xs) = f x : map f xs
map : forall (k : Clock) a b. (a -> b) -> Stream k a -> Stream k b.
map = \f ->
  let mapfix = \g xs ->
        case unfold xs of
        | Cons x xs' ->
          let ys = \\(af : k) -> g [af] (xs' [af])
          in  cons (f x) ys
  in fix mapfix.

-- in Haskell:
-- maap f (x1 : x2 : xs) = f x1 : f x2 : maap f xs
maapfix : forall (k : Clock) a b. (a -> b) -> |>k (CoStream a -> CoStream b) -> CoStream a -> CoStream b.
maapfix = \f r xs ->
  let h = hd xs in
  let t = tl xs in
  let h' = hd t in
  let t' = tl t in
  let inner = \r' -> cos (f h') (pure (r' t'))
  in  cos (f h) (dmap inner r).

maap : forall a b. (a -> b) -> CoStream a -> CoStream b.
maap = \f -> fix (maapfix f).
```

Listing 3: `map` and `maap` from [3] encoded in CloFRP

In Haskell, `map` and `maap` have the same type, since their behaviour is, at first glance, identical. `map` is just normal stream/list map, and `maap` does the same, just two elements at a time instead of one. Both definitions have the same behaviour when applied to fully constructed streams, *but not when applied to streams that are "still being constructed"*. Consider these two Haskell definitions of a stream of natural numbers:

```
nats = 0 : (map (\x -> x + 1) nats)
badnats = 0 : (maap (\x -> x + 1) badnats)
```

`nats` is fine but `badnats` will not produce anything as `maap` expects two elements to be available on the input stream, but it is only ever given one. This difference in behaviour is subtle and not expressible in Haskell's type-system. However, looking at the types given in their CloFRP definitions, they clearly reflect that `maap` will only work on `CoStream` which represents "fully-constructed" streams, as they are productive for any clock. Thus, `badnats` will not type-check in CloFRP.

5.4.2 Monads

We can also write higher-kinded abstractions in CloFRP. Listing 4 shows that we can model monads in CloFRP quite easily.

```
data Monad (m : * -> *) =
  Monad (forall a. a -> m a) (forall a b. (a -> m b) -> m a -> m b).
  --      ^^^^^^^pure^^^^^^^  ^^^^^^^^^^^^^^^^bind^^^^^^^^^^^^^^^^^
maybeM : Monad Maybe.
maybeM = Monad
  Just
  (\fn x -> case x of
    | Nothing -> Nothing
    | Just x' -> fn x'
  ).
```

Listing 4: Monads and an instance for `Maybe` in CloFRP.

While monads and other higher-kinded types are expressible in CloFRP, their practical use is limited in the absense of a proper type-class system, but this example shows that the foundation of the type-system scales well.

5.4.3 Circular traversal of binary trees

As Atkey and McBride show in [3], clock quantification can also be used to encode some of the strange circular programs that can be constructed in Haskell using laziness. A classic example, due to Bird [36], is the act of replacing all values in a binary tree with its minimum using only a single pass. We can construct such a procedure in Haskell as shown in Listing 5.

```
replaceMin :: Tree Int -> Tree Int
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', ml) = replaceMinBody l m
        (r', mr) = replaceMinBody r m
    in (Br l' r', min ml mr)
```

Listing 5: Haskell program that replaces all values in a binary tree with its minimum in one pass.

The first line is the interesting one, since `let (t', m) = replaceMinBody t m in t'` both uses and binds `m` in the same let-binding, which only works thanks to Haskell's lazy nature.

Atkey and McBride encode this program in [3] by replacing the circular let with a *feedback* combinator typed as $\forall\kappa.(\triangleright^\kappa U \to (B[\kappa], U)) \to B[k]$ in their calculus. Here, $B[\kappa]$ signifies that κ is free in B. B and U are meta-variables for types here, as their calculus does not have polymorphism. We can encode this concept in CloFRP by defining a data-type `Delay` that models types parameterized over clocks. `feedback` can then be encoded as:

```
data Delay a (k : Clock) = Delay (|>k a). -- flip arguments to |> and promote it to type-constructor
-- mapping over |>k
dmap : forall (k : Clock) a b. (a -> b) -> |>k a -> |>k b.
dmap = \f la -> (\\(af : k) -> f (la [af])).

feedback : forall (k : Clock) (b : Clock -> *) u. (|>k u -> (b k, u)) -> b k.
feedback = \f -> fst (fix (\x -> f (dmap snd x))). -- x has type |>k (b k, u)
```

Here we can leverage the expressive power that parametric polymorphism and type-constructors gives us. We can then translate their encoding of `replaceMin` almost ad verbatim and define a program we can execute (Listing 6).

```
data TreeF a f = Leaf a | Br f f deriving Functor.
type Tree a = Fix (TreeF a).

-- |>k is an applicative functor
app : forall (k : Clock) a b. |>k (a -> b) -> |>k a -> |>k b.
app = \lf la -> \\(af : k) ->
  let f = lf [af] in
  let a = la [af] in
  f a.

replaceMinBody : forall (k : Clock). Tree Int -> |>k Int -> (Delay (Tree Int) k, Int).
replaceMinBody = primRec {TreeF Int} (\t m ->
  case t of
  | Leaf x -> (Delay (map leaf m), x)
  | Br (l, lrec) (r, rrec) ->
    let (Delay l', ml) = lrec m in
    let (Delay r', mr) = rrec m in
    let m'             = min ml mr
    in  (Delay (app (map br l') r'), m')
).

replaceMinK : forall (k : Clock). Tree Int -> Delay (Tree Int) k.
replaceMinK = \t -> feedback (replaceMinBody t).

replaceMin : Tree Int -> Tree Int.
replaceMin = \t ->
  let Delay t' = replaceMinK {K0} t
  in t' [<>].

ofHeight : Nat -> Tree Int.
ofHeight = \nat ->
  fst (primRec {NatF} (\m n ->
    case m of
      | Z -> (leaf n, 1 + n)
      | S (m', r) ->
        let (t1, n1) = r n in
        let (t2, n2) = r n1
        in  (br t1 t2, n2)
  ) nat 0).

main : Tree Int.
main =
  let five = s (s (s (s z))) in
  let four = s (s (s z)))
  in  replaceMin (ofHeight (plus four five)).
```

Listing 6: CloFRP program that replaces all values in a binary tree of depth 9 with its minimum in one pass.

5.4.4 Stream processing

As Atkey and McBride show in [3], we can use guarded recursion to encode "Stream Processors" as defined by Ghani, Hancock and Pattinson [37]. Atkey and McBride use nested fixpoint types, but CloFRP does not have nested fixpoints, as recursive types are represented by a type-constructor instead of a recursive substitution. However, we can formulate stream processors in the style of [38] by using algebraic data-types as seen in Listing 7.

```
data SPF i o (k : Clock) f
  = Get (i -> f)
  | Put o (|>k f)
  deriving Functor.

type SP i o (k : Clock) = Fix (SPF i o k).
apply : forall (k : Clock) i o. SP i o k -> CoStream i -> CoStream o.
apply = fix (\rec ->
  primRec {SPF i o k} (\x s ->
    case x of
    | Get f -> (snd (f (hd s))) (tl s)
    | Put b sp ->
      let sp1 = dmap fst sp in
      cos b (app (app rec sp1) (pure s))
  )).

spid : forall i. SP i i K0.
spid = fix (\f -> fold (Get (\i -> fold (Put i f)))).
```

Listing 7: Stream processors in the style of Danielsson and Altenkirch [38].

Stream processors are constructed by either `Get` which gets a value from the input stream, or `Put` which puts a value to the output stream. `SPF` is parameterized over the type of inputs `i` and outputs `o` and a clock-variable `k`. Notice the functorial type-parameter `f`; in `Get` it appears unguarded, while in `Put` it is guarded behind a $\triangleright^{\kappa}$. This means that a processor can only read a finite number of elements before producing at least one element of output.

The semantics of stream processors are encoded by the `apply` function, which defines how a processor transforms an input stream to an output stream. Notice that `apply` is defined using primitive recursion *inside* an outer fixpoint. Using the applicative functor combinators `app`, `dmap` and `pure`, which we defined earlier, makes the definition quite concise. This example shows that CloFRP can express safe coinductive types and definitions much in the same ways as Agda, but entirely without a separate productivity checker.

Finally, `spid` is the simplest stream processor that returns its input stream un-altered.

5.4.5 Impredicative polymorphism

We can instantiate polymorphic definitions with polymorphic types using explicit type-applications, shown in Listing 8.

```
id : forall a. a -> a.
id = \x -> x.

pred : forall a. a -> a.
pred = id id. -- without explicit application we get a predicative instantiation

imp : (forall a. a -> a) -> (forall a. a -> a).
imp = id {forall a. a -> a}. -- with an annotation, we can have impredicativity

data Maybe a = Nothing | Just a.

-- we can instantiate type-constructors with poly-types
impM : Maybe (forall a. a -> a).
impM = Just {forall a. a -> a} id.

-- nothing special here
default : forall a. a -> Maybe a -> a.
default = \def m ->
  case m of
  | Nothing -> def
  | Just x -> x.

imp2 : Maybe (forall a. a -> a) -> forall a. a -> a.
imp2 = \x -> default id x.

imp2eta : Maybe (forall a. a -> a) -> forall a. a -> a.
imp2eta = default {forall a. a -> a} id.
```

Listing 8: Impredicative polymorphism in CloFRP.

`imp` is a classical example of impredicativity, as `id` is instantiated with its own type signature. If the explicit application is left out, we get the predicative instantiation as witnessed by `pred`. `imp2` shows that we can work with impredicatively instantiated types in a natural way, but there is a catch: `imp2eta` is η-equivalent to `imp2` but will not type-check without the explicit type-application. Since `imp2eta` is defined by an application, the only rule to use when type-checking it is subsumption (Sub), which will infer the type `(Maybe (∃`d -> ∃`d)) -> ∃`d -> ∃`d` where `∃`d` is an existential type-variable that has been machine-generated. Subsumption will then switch to the subtyping judgment `(Maybe (∃`d -> ∃`d)) -> ∃`d -> ∃`d <: (Maybe (∀a. a -> a)) -> ∀a. a -> a`, which will fail when checking the codomain as `∃`d -> ∃`d <: ∀a. a -> a` is not true. This shows that, as noted in [30], impredicativity interacts with quantifiers in a complicated way which compromises typability under η-equivalence.

We do not have a good story for pattern-matching on values of impredicative types. Listing 9 shows a problematic definition.

```
data Wrap a = MkWrap a.
data A = A.
foo : Wrap (forall a. a -> a) -> A.
foo = \w ->
  case w of
  | MkWrap id -> A. -- FAILS!
```

Listing 9: A program that shows that pattern-matching and impredicative polymorphism does not work together.

This definition requires instantiating an existential type-variable with a polymorphic type, which is not possible with our inference system. Specifically, the full error message given will be:

```
( Other error: Assigning `d to `c
Caused by: Context is not well-formed due to ^`d = `c
Caused by: Other error: kindOf `c
Caused by: Other error: queryKind: Cannot lookup kind of `c in [w λ: Wrap (∀a. a -> a) , †`a , ^`e]
, [w λ: Wrap (∀a. a -> a) , †`a , ^`e , ^`d , ^`b = ∃`d -> ∃`e , `c] )
Progress:
============ foo =============
[->I] \w -> case w of | MkWrap id -> A <= (Wrap (∀a. a -> a)) -> A     in []
  [Case<=] case w of | MkWrap id -> A <= A
    [Var] w => Wrap (∀a. a -> a)
    [Intros] Wrap (∀a. a -> a)
    [CheckClause] | MkWrap id -> A <= A     in [w λ: Wrap (∀a. a -> a) , †`a]
      [CheckPat] MkWrap id <= Wrap (∀a. a -> a)     in [w λ: Wrap (∀a. a -> a) , †`a]
        [<:TApp] Wrap ∃`b <: Wrap (∀a. a -> a)     in [w λ: Wrap (∀a. a -> a) , †`a , ^`b]
          [<:Free] Wrap <: Wrap
          [<:∀R] ∃`b <: ∀a. a -> a     in [w λ: Wrap (∀a. a -> a) , †`a , ^`b]
            [InstantiateL] ^`b :<= `c -> `c
              [InstLArr] `b :<= `c -> `c     in [w λ: Wrap (∀a. a -> a) , †`a , ^`b , `c]
                [InstRError] ^`d = `c     in [ w λ: Wrap (∀a. a -> a) , †`a , ^`e , ^`d
                                               , ^`b = ∃`d -> ∃`e , `cI
                                               ]
```

The error message above shows that the problem originates in `<:∀R`, as it decomposes the polymorphic type instead of instantiating directly. Therefore, the existential `^`d` is assigned to the universal variable `` `c `` which is *to the right* of `^`d` in the context, and therefore not a valid assignment. This is by design, so we cannot simply circumvent it. But in the future we could introduce new syntax to let the programmer guide the type-checker in these cases, such as by annotating patterns with types.

This concludes the section on typing example programs in CloFRP.

6 Operational semantics of CloFRP

The operational semantics of CloFRP are derived from the reduction semantics specified for CloTT in [4], and which can be seen in Figure 10. Since CloFRP is simply-typed and lacks the delayed fixpoint dfix in favor of a non-delayed version, there are some slight changes in handling fixpoints. The same is true for fold and unfold, which are no longer parameterized over ticks. Furthermore, CloFRP is a significantly larger language and also has primitive recursion and user-defined types, and thus also user-defined values, namely constructors. Luckily, none of these features significantly complicate the operational semantics, but they are important nonetheless – in particular, primitive recursion relies on the fmap primitive which in turn relies on a simple form of dynamic dispatch, since it is overloaded.

The operational semantics consists of three main relations: a *one-step* relation $\Xi \vdash e \Downarrow v$ which is a mostly-standard call-by-value big-step semantics, a *force* relation $\Xi \vdash v \lightning v'$, which forces all tick-abstractions to be evaluated, and finally a *coinductive tick* relation $\Xi \vdash v \ggg v'$ wich combines the two previous relations. In this sense, it is somewhat similar to the operational semantics defined in [2], however, our language does not enjoy the guarantees regarding space- and time-leaks of that work. On the other hand, our language is more expressive in the sense that it can model acausal corecursion[2].

6.1 Values and contexts

Values represent normal forms of the terms in CloFRP. Figure 32 describes the syntax of values and evaluation contexts.

[2] and primitive recursion, but that is less novel in this context.

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & \alpha \mid \diamond \mid \lambda_{\Xi}\, x.\ e \mid \gamma_{\Xi}\, \alpha.\ e \mid \langle v_1, \ldots, v_n \rangle \mid \mathcal{C}\, v_1, \ldots, v_2 \mid \mathsf{fold}\, v \\
\text{Evaluation contexts} & \Xi & ::= & \cdot \mid \Xi, x \mapsto v, fmap(F) \mapsto v
\end{array}
$$

Figure 32: Values and evaluation contexts in CloFRP.

Values are ticks and tick-variables, closures (lambdas with a bound context), tick-closures, tuples of values, and finally constructors $\mathcal{C}$ with a list of fields. Lambda closures $\lambda_{\Xi}\, x.\ e$ is a lambda that closes over an environment Ξ. Tick closures can be understood similarly. Finally, evaluation contexts Ξ simply map names to values, and type-constructors to their corresponding functorial mapping, which we assume to have been generated in a pre-processing step. Evaluation contexts are stack-like, in the way that if there are multiple mappings of a name to a value, the right-most mapping is selected in the VAR rule.

6.2 One-step semantics

The one-step semantics can be seen in Figure 33. The main relation $\Xi \vdash e \Downarrow v$ evaluates an expression e in CloFRP to a value v under an evaluation context Ξ. The pattern-matching semantics $\Xi \vdash \rho \downharpoonright v \dashv \Xi'$ proceeds inductively over the form of the pattern and binds the relevant values in the output context Ξ'. The operational semantics are formulated in a style without direct subtitutions on values. Instead, name-bindings are kept in the context Ξ. This is done to keep the specification as close to the implementation as possible, without introducing unnecessary overhead.

$\boxed{\Xi \vdash e \Downarrow v}$ Under evaluation context Ξ, e evaluates to v in one step

$\boxed{\Xi \vdash \rho \downarrow v \dashv \Xi'}$ Under evaluation context Ξ, matching v with ρ yields context Ξ'

$$\frac{}{\Xi \vdash \diamond \Downarrow \diamond}\ \textsc{Tick} \qquad \frac{}{\Xi \vdash \alpha \Downarrow \alpha}\ \textsc{TickVar} \qquad \frac{(x \mapsto v) \in \Xi}{\Xi \vdash x \Downarrow v}\ \textsc{Var}$$

$$\frac{}{\Xi \vdash \lambda x.\ e \Downarrow \lambda_{\Xi} x.\ e}\ \textsc{Abs} \qquad \frac{}{\Xi \vdash \gamma(\alpha : \mathsf{c}).\ e \Downarrow \gamma_{\Xi}\, \alpha.\ e}\ \textsc{TickAbs} \qquad \frac{(fmap(F) \mapsto v) \in \Xi}{\Xi \vdash \mathsf{fmap}_F \Downarrow v}\ \textsc{Fmap}$$

$$\frac{\Xi \vdash e \Downarrow v}{\Xi \vdash e : A \Downarrow v}\ \textsc{Ann} \qquad \frac{\Xi \vdash e \Downarrow v}{\Xi \vdash e\,\{A\} \Downarrow v}\ \textsc{TypeApp} \qquad \frac{\Xi \vdash e_1 \Downarrow v_1 \ \ldots\ \Xi \vdash e_n \Downarrow v_n}{\Xi \vdash \langle e_1, \ldots, e_n \rangle \Downarrow \langle v_1, \ldots, v_n \rangle}\ \textsc{Tuple}$$

$$\frac{\Xi \vdash e' \Downarrow v' \qquad \Xi \vdash \rho \downarrow v' \dashv \Xi' \qquad \Xi' \vdash e \Downarrow v}{\Xi \vdash \mathsf{let}\ \rho\ = e'\ \mathsf{in}\ e \Downarrow v}\ \textsc{Let}$$

$$\frac{\Xi \vdash e \Downarrow v \qquad \text{Smallest } i \text{ such that } \Xi \vdash \rho_i \downarrow v \dashv \Xi' \qquad \Xi' \vdash e_i \Downarrow v_i}{\Xi \vdash \mathsf{case}\ e\ \mathsf{of}\ (\rho_1 \longrightarrow e_1, \ldots, \rho_n \longrightarrow e_n) \Downarrow v_i}\ \textsc{Case} \qquad \frac{\Xi \vdash e \Downarrow v}{\Xi \vdash \mathsf{fold}\, e \Downarrow \mathsf{fold}\, v}\ \textsc{Fold}$$

$$\frac{\Xi \vdash e \Downarrow \mathsf{fold}\, v}{\Xi \vdash \mathsf{unfold}\, e \Downarrow v}\ \textsc{Unfold} \qquad \frac{\Xi \vdash \lambda f.\ f\ (\gamma(\_ : \kappa).\ \mathsf{fix}\, f) \Downarrow v}{\Xi \vdash \mathsf{fix} \Downarrow v}\ \textsc{Fix}$$

$$\frac{\Xi \vdash \lambda b\, o.\ b\,(\mathsf{fmap}_F\ (\lambda i.\ \langle i, \mathsf{primRec}_F\ b\ i \rangle)\ (\mathsf{unfold}\ o)) \Downarrow v}{\Xi \vdash \mathsf{primRec}_F \Downarrow v}\ \textsc{PrimRec}$$

$$\frac{\Xi \vdash e_1 \Downarrow \lambda_{\Xi'} x.\ e \qquad \Xi \vdash e_2 \Downarrow v_2 \qquad \Xi', x \mapsto v_2 \vdash e \Downarrow v}{\Xi \vdash e_1\, e_2 \Downarrow v}\ \textsc{LamApp}$$

$$\frac{\Xi \vdash e_1 \Downarrow \gamma_{\Xi'} x.\ e \qquad \Xi \vdash e_2 \Downarrow v_2 \qquad \Xi', x \mapsto v_2 \vdash e \Downarrow v}{\Xi \vdash e_1\,[e_2] \Downarrow v}\ \textsc{TickApp}$$

$$\frac{\Xi \vdash e_1 \Downarrow (\mathcal{C}\, v_1, \ldots, v_n) \qquad \Xi \vdash e_2 \Downarrow v}{\Xi \vdash e_1\, e_2 \Downarrow (\mathcal{C}\, v_1, \ldots, v_n, v)}\ \textsc{ConstrApp}$$

$$\frac{}{\Xi \vdash x \downarrow v \dashv \Xi, x \mapsto v}\ \textsc{PatBind} \qquad \frac{\Xi \vdash \rho_1 \downarrow v_1 \dashv \Xi_1 \ \ldots\ \Xi_{n-1} \vdash \rho_n \downarrow v_n \dashv \Xi_n}{\Xi \vdash \langle \rho_1, \ldots, \rho_n \rangle \downarrow \langle v_1, \ldots, v_n \rangle \dashv \Xi_n}\ \textsc{PatTuple}$$

$$\frac{\Xi \vdash \rho_1 \downarrow v_1 \dashv \Xi_1 \ \ldots\ \Xi_{n-1} \vdash \rho_n \downarrow v_n \dashv \Xi_n}{\Xi \vdash (\mathcal{C}\, \rho_1, \ldots, \rho_n) \downarrow (\mathcal{C}\, v_1, \ldots, v_n) \dashv \Xi_n}\ \textsc{PatConstr}$$

Figure 33: Big-step operational one-step-semantics of CloFRP.

Most of the rules are quite straight-forward. Tick and TickVar says that the tick-constant simply evaluates back to itself, and the same is true for tick-variables. Normal variable names are looked up in the context (Var). Lambda- and tick-abstractions are evaluated to their closure form (Abs,TickAbs). Evaluating fmap_F results in the definition of fmap for F, which we assume has been elaborated in a pre-processing step. Annotated expressions (Ann) are simply evaluated by ignoring their annotations. The same principle applies to explicit type-applications (TApp). Tuples (Tuple) are evaluated left-to-right (could equally be right-to-left). Let-bindings (Let) use the pattern-matching semantics to bind e' to a pattern ρ before evaluating the body e.

The recursive primitives are somewhat more interesting. As described in [4], fix is equivalent to the delayed fixpoint dfix^{κ} unfolded once immediately. Therefore, $\mathsf{fix}\, f = f(\mathsf{dfix}^{\kappa} f)$. Since we do not have dfix^{κ} in our syntax, we simulate it by delaying f by wrapping it in a tick-abstraction. The abstracted tick is never used in the body, and neither is the clock-variable important during evaluation, so these can

be arbitrary. We use _ to signify a name that cannot be used, and thus cannot be free in f.

Primitive recursion (PrimRec) is slightly more involved, but follows more or less exactly from how one would implement primitive recursion in terms of general recursion over functors. $\mathsf{primRec}_F$ evaluates to a function that takes a body b and some concrete member of the functor F, bound to o, and applies the body b structurally to o by using fmap_F to "go under" the structure of o. The catch is that it also keeps a copy of o around at each iteration, so that b can use both the "current" o along with the result of the recursion. When evaluating a primitive-recursive definition, the entire definition is eagerly evaluated before the evaluation proceeds.

Application of lambda abstractions and tick-abstractions (LamApp,TickApp) follow the same general pattern. Note that there is no distinction between applying a tick-abstraction with a tick-variable or a tick-constant. Since we do not evaluate under tick-abstractions, this is not necessary, as all tick-variables must have been bound to the tick-constant before evaluation of the term happens.

The rule ConstrApp models application of constructors to values. The applied value is simply appended to the fields of the constructor $\mathcal{C}$. This quite elegantly allows for partial application of constructors to be modeled. Note that this rule has the same syntactic form in the conclusion as LamApp. Luckily, selecting between the two rules is easily done, as the first premise serves as a guard which uniquely determines what rule to select (basically a side-condition).

The pattern-matching semantics $\Xi \vdash \rho \lfloor v \dashv \Xi'$ is a pretty straightforward relation that simply proceeds inductively on the shape of the pattern and the value to be matched against. Each leaf of the derivation binds a name in PatBind.

6.3 Coinductive tick-semantics

The one-step semantics given in 6.2 will only evaluate a CloFRP program one step – it stops when reaching any tick-abstractions, since it cannot continue until another "tick on the clock happens". To simulate the tick on any such clock, we formulate the coinductive "tick-semantics" that will keep evaluating a possibly non-terminating program. The tick-semantics relation can be seen in Figure 34.

$\boxed{\Xi \vdash v \ggg v'}$ Under evaluation context Ξ, v evalues to v' in the next tick

$\boxed{\Xi \vdash v \lightning v'}$ Under evaluation context Ξ, force v to v'

$$\frac{\Xi \vdash v_1 \lightning v_1' \ \ldots \ \Xi \vdash v_n \lightning v_n' \qquad \Xi \vdash v_1' \ggg v_1'' \ \ldots \ \Xi \vdash v_n' \ggg v_n''}{\Xi \vdash (\mathcal{C}\, v_1, \ldots, v_n) \ggg (\mathcal{C}\, v_1'', \ldots, v_n'')} \ggg\textsc{Constr}$$

$$\frac{\Xi \vdash v_1 \lightning v_1' \ \ldots \ \Xi \vdash v_n \lightning v_n' \qquad \Xi \vdash v_1' \ggg v_1'' \ \ldots \ \Xi \vdash v_n' \ggg v_n''}{\Xi \vdash \langle v_1, \ldots, v_n \rangle \ggg \langle v_1'', \ldots, v_n'' \rangle} \ggg\textsc{Tuple} \qquad \frac{\Xi' \vdash v \ggg v'}{\Xi \vdash \mathsf{fold}\, v \ggg \mathsf{fold}\, v'} \ggg\textsc{Fold}$$

$$\frac{}{\Xi \vdash v \ggg v} \ggg\textsc{Value} \qquad \frac{\Xi', \alpha \mapsto \diamond \vdash e \Downarrow v}{\Xi \vdash \gamma_{\Xi'}\, \alpha.\ e \lightning v} \lightning\textsc{TickAbs} \qquad \frac{v \neq \gamma_{\Xi'}\, \alpha.\ e}{\Xi \vdash v \lightning v} \lightning\textsc{Value}$$

Figure 34: Tick-semantics for CloFRP.

The tick-semantics is quite simple. $\Xi \vdash v \ggg v'$ just proceeds structurally over values. If it encounters primitives, it just returns them (≫Value). When it goes under a constructor (≫Constr), or a tuple (≫Tuple), it attempts to force any delayed expressions under them. Since we only have guarded recursion to worry about at this point, this is totally safe. Values fold must have constructors v directly under them, so we simply proceed into v (rule ≫Fold). Rule ↯TickAbs says that to force a tick-abstraction, we simply assign the tick-constant $\diamond$ to the bound tick-variable and do a one-step evaluation of the body of the abstraction. Finally, rule ↯Value says that all other values simply force to themselves. Importantly, the $\Xi \vdash v \ggg v'$ relation does not necessarily terminate, but it itself uses guarded recursion, in the sense that every inductive step is guarded under a value-constructor, so it is productive when evaluated over productive terms in CloFRP, which all terms should be if they type-check.

Finally, to evaluate a program e in CloFRP, one can simply first evaluate e to v with the one-step semantics $\Xi \vdash e \Downarrow v$ and then apply the recursive tick-semantics $\Xi \vdash v \ggg v'$ to keep evaluating the expression corecursively.

This concludes the section on the operational semantics of CloFRP.

7 Implementation

Rather than implementing CloFRP as a standalone language, we set out to integrate it tightly with Haskell. Using Haskell's `QuasiQuote` mechanism, one can embed an entirely separate language within Haskell. A (somewhat) type-safe integration with Haskell can then be achieved by using Haskell's constructs for type-level programming. Integrating a custom language with Haskell provides several benefits, for example one can choose to implement selected parts of one's application in a language that fits the domain model (FRP in our case) and then piggy-back on Haskell to provide extra libraries and input/output.

7.1 Overview

The implementation consists of a Haskell library for parsing, type-checking and running CloFRP programs. Furthermore, there are a few example programs, just-over 310 hand-written scenario tests and some benchmarks (mainly aimed at profiling the implementation). The library is just over 3600 lines of Haskell code (without comments), while the test-suite is 3700 lines of code.

The library is split into several folders and modules, all defined within the `CloFRP` namespace:

- `AST` contains the abstract syntax tree of programs written in CloFRP: expressions, types, patterns, names, primitives, declarations and programs.
- `Check` contains code to type-check and elaborate programs written in CloFRP. The meat of the implementation lies within this namespace.
- `Eval` contains code that evaluates CloFRP programs.
- `Parser` contains parsers for various CloFRP terms.
- `Derive` contains code to derive functors from data-type declarations.
- `Interop` defines how to combine CloFRP programs with Haskell programs in a somewhat typesafe manner.
- `QuasiQuoter` defines quasi-quoters that allow Haskell programmers to write programs in CloFRP easily.
- `Annotated, Context, Pretty` and `Utils` contain mostly un-interesting helper functions to work with annotated AST's, different contexts, pretty-printing ASTs and assorted utility functions.

Parsing is mostly uninteresting, so we shall not go through the code that parses CloFRP programs. However, we shall touch briefly on the rest of the implementation. There is quite a bit of code, so a thorough walkthrough of the implementation is beyond the scope of this report. Instead, we shall focus on the high-level strokes of the implementation in terms of data-types and core functions. The implementation can be found at https://github.com/adamschoenemann/clofrp.

7.2 Abstract Syntax Trees

There are several different ASTs in CloFRP: types and kinds, expressions and patterns, and programs and declarations.

7.2.1 Types and kinds

Types in CloFRP are represented by a generalized algebraic data-type shown in Listing 10. Note that it uses a feature of the GHC (Glasgow Haskell Compiled) extension `DataKinds` that allows us to promote data-constructors to the kind-level. Thus, the declaration of `TySort` gives rise not only to the type `TySort` with two nullary data-constructors `Mono` and `Poly`, but also to the *kind* `TySort` with member-types `Mono` and `Poly`. Expressing the type for CloFRP types as a GADT then allows us to specify that the second type-parameter to `Type'` must have kind `TySort`. This allows us to distinguish at the type-level whether a type is mono- or poly-type.

```
type Type a s = Annotated a (Type' a s)

data TySort = Mono | Poly deriving (Show, Eq)

data Type' :: * -> TySort -> * where
  TFree   :: Name                          -> Type' a s     -- Foo
  TVar    :: Name                          -> Type' a s     -- x
  TExists :: Name                          -> Type' a s     -- α^
  TApp    :: Type a s     -> Type a s      -> Type' a s     -- F B
  (:->:)  :: Type a s     -> Type a s      -> Type' a s     -- A -> B
  Forall  :: Name -> Kind -> Type a 'Poly  -> Type' a 'Poly -- ∀(α : χ). A
  RecTy   :: Type a s                      -> Type' a s     -- Fix F
  TTuple  :: [Type a s]                    -> Type' a s     -- (A₁,...,Aₙ)
  Later   :: Type a s     -> Type a s      -> Type' a s     -- |>k A

type PolyType a = Type a 'Poly
type MonoType a = Type a 'Mono

data Kind = Star | ClockK | Kind :->*: Kind
data Name = UName String | MName Integer | DeBruijn Integer
```

Listing 10: The AST data-type for CloFRP types, kinds and names.

Using the `Annotated` data-type is a pattern we shall see in many places throughout the code-base. `Annotated` is simply defined as `data Annotated a f = A a f`, so all it does is add an annotation to `a` type `f`. Notice that the two types `Type` and `Type'` are defined mutually-recursively. This pattern allows us to have alternating values of `Type` and `Type'` as we go down the AST of types, and thus there will be an annotation at every level. This annotation can be any type `a`, but will in most cases be a position in the source from which the type is parsed.

Annotating values with other values elegantly is not that easy in Haskell, and the solution used above is one of many. Another particularly elegant solution is to use recursion schemes with the `Cofree` type. Using recursion schemes provides many benefits, and you can get many definitions "for free". On the other hand, it can limit the expressibility and clarity of your code quite a bit, as you have to use the weirdly-named combinators such as cata-, para-, hylo- and zygohisto-morphisms to express recursion. The pattern we have used here is less elegant, and we are forced to write many definitions out by hand, and to take care of the annotated values all the time, but it wins on simplicity and flexibility.

Once you are used to reading GADT definitions, it should be clear that the above declaration corresponds almost exactly to the syntax defined in Figure 20.

An important aside is the signature for `Forall`. In the code, it explicitly takes a poly-type argument, whereas in theory a forall could easily wrap a mono-type. However, this is necessary to avoid introducing existential types in Haskell, which significantly complicates certain definitions. If we replaced `'Poly` with a type-variable `s` in the thrid argument of `Forall`, then pattern-matching with the `Forall` pattern would introduce an existential type in the context, since we cannot know from the type of the scrutinee what the type of the argument was! There are ways to work with this in Haskell, but they are somewhat complicated. In addition, existential types break many nice things, like the automatic deriving of many

type-class instances, so we take care to avoid them here. In practice, this is not a problem, since all mono-types are also poly-types, so if we wish to create a `Forall` type that wraps a mono-type τ we can simply upcast τ to a polytype.

Kinds are the inhabited kind ($\star$), the kind of clocks and the arrow combinator for kinds. Syntactic well-formedness of kinds as discussed previously is taken care of by a validation function.

Finally, names come in three varieties: user-defined names `UName` are "normal" names typically introduced when parsing concrete syntax, machine-names `MName` which are freshly generated names during type-checking, and DeBruijn indices (which are only really used during expansion of type-synonyms).

7.2.2 Expressions and patterns

A CloFRP expression is parsed, typed and evaluated from a single type that represents its abstract syntax, shown in Listing 11.

```
type Expr a = Annotated a (Expr' a)
data Expr' a
  = Var Name                              -- x
  | TickVar Name                          -- [x]
  | Ann (Expr a) (PolyType a)             -- e : A
  | App (Expr a) (Expr a)                 -- e1 e2
  | Lam Name (Maybe (PolyType a)) (Expr a) -- \x -> e OR \(x : A) -> e
  | TickAbs Name Name (Expr a)            -- \\(α : κ) -> e
  | Tuple [Expr a]                        -- n-ary tuples
  | Let (Pat a) (Expr a) (Expr a)         -- let p = e1 in e2
  | Case (Expr a) [(Pat a, Expr a)]       -- case e of | p₁ -> e₁ | ... | pₙ -> eₙ
  | TypeApp (Expr a) (PolyType a)         -- e {A}
  | Fmap (PolyType a)                     -- fmap_F
  | PrimRec (PolyType a)                  -- primRec_F
  | Prim P.Prim                           -- primitives are ints, (un)fold, fix, primRec etc

type Pat a = Annotated a (Pat' a)
data Pat' a
  = Bind Name           -- x
  | Match Name [Pat a]  -- P ρ₁,...,ρ₂
  | PTuple [Pat a]      -- (ρ₁,...,ρ₂)
```

Listing 11: The AST data-type for CloFRP expressions and patterns.

Again, the constructors correspond closely with the syntax specification. Furthermore, expressions use the same pattern of using mutual recursion with the `Annotated` type to model arbitrary annotations. Lambdas have optional type-signatures, which is modeled by using the `Maybe` type to wrap the signature. The most interesting departure from the specification is how to handle tick-application. In the specification, there are two kinds of application: $e_1\, e_2$ which is normal application that eliminates arrow types; and $e\,[\alpha]$ which is application of delayed expressions with ticks, which eliminates later-types $\triangleright^\kappa A$. However, this syntax is just bothersome to parse, as they both rely on the application-operator " " (space). We could change the syntax, but instead we have opted to encode tick-application by having a separate constructor for tick-variables `TickVar`. A tick-application `e [alpha]` is then parsed as `e1 `App` (TickVar alpha)`. This also makes tick-variables slightly more first-class, so you can alias them in let-bindings for example. The consequence is that almost all code that deals with application will need to take two cases into account.

Let-bindings and case-clauses can contain patterns, which are either name-bindings, constructor-patterns `Match`, or tuple-patterns. Patterns are also encoded with `Annotated`.

7.2.3 Programs and declarations

Listing 12 show the data-types used to define CloFRP programs.

```haskell
data Prog a = Prog [Decl a]

type Decl a = Annotated a (Decl' a)
data Decl' a
  = FunD Name (Expr a)      -- function definitions
  | DataD (Datatype a)      -- data declarations
  | SigD Name (PolyType a)  -- signatures
  | SynonymD (Synonym a)    -- type-synonyms

data Datatype a = Datatype { dtName     :: Name
                           , dtBound    :: [(Name, Kind)]
                           , dtConstrs  :: [Constr a]
                           , dtDeriving :: [String]
                           }
type Constr a = Annotated a (Constr' a)
data Constr' a = Constr Name [PolyType a]

data Synonym a = Synonym { synName      :: Name
                         , synBound     :: [(Name, Kind)]
                         , synExpansion :: PolyType a
                         }
```

Listing 12: The AST data-type for CloFRP programs and declarations.

A program is simply a list of declarations. A declaration can be one of four things: 1) a function definition `x = e` that binds an expression to a name 2) a data-type declaration 3) a type-signature `x : A`, or 4) a type-synonym.

Data-types consists of a name, a list of bound type-variables and their kinds, a list of constructors, and a list of deriving clauses. Currently, the only thing a data-type can derive is "Functor", but this declaration allows for more derivable classes in the future.

A constructor is simply a name along with a list of types that signify the type of its arguments. For example, `data Maybe a = Nothing | Just a deriving Functor` is encoded as:

```haskell
Datatype
  { dtName = "Maybe"
  , dtBound = [("a", Star)]
  , dtConstrs =
      [ Constr "Nothing" []
      , Constr "Just" [TVar "a"]
      ]
  , dtDeriving = ["Functor"]
  }
```

Type-synonyms are similarly encoded with a name, a list of bound type-variables, and the type the synonym expands to.

7.3 Type-inference

Type-inference is by far the largest part of the implementation, and also in many ways the most complex. The code follows the algorithmic specification given in section 5.3 quite closely, but obviously needs to implement lots of the smaller definitions that are glossed over in the specification, and has to do some extra book-keeping to prevent name-capture and report (somewhat helpful) errors.

7.3.1 Local contexts

Local contexts are modeled as shown in Listing 13.

```
data CtxElem a
  = Uni Name Kind                       -- Universal type-variables
  | Exists Name Kind                    -- Existential type-variables
  | (Binding, Name) `HasType` PolyType a -- x :ᵤ A
  | Name := MonoType a                  -- α^ = τ
  | Marker Name                         -- |>α^

data Binding = LamB | LetB
newtype LocalCtx a = LocalCtx { unGamma :: [CtxElem a] }

infixl 5 <+
(<+) :: LocalCtx a -> CtxElem a -> LocalCtx a
LocalCtx xs <+ x = LocalCtx (x : xs)

infixl 4 <++
(<++) :: LocalCtx a -> LocalCtx a -> LocalCtx a
LocalCtx xs <++ LocalCtx ys = LocalCtx (ys ++ xs)

instance Monoid (LocalCtx a) where
  mempty = LocalCtx []
  mappend = (<++)
```

Listing 13: Local typing contexts.

A local typing context is just a list of context elements. The Haskell `newtype` declaration is just like a regular `data` declaration, but can only have one constructor. The runtime representation of the newtype will be the same as its definition. Context elements correspond with the syntax given in Figure 24. There are universal type-variables, existentials, assertions that names have types (bound through a lambda or a let-binding), solutions to existentials, and markers.

Haskell-lists are left-prepend lists, while contexts in our presentation are right-append. In practice it does not really matter, but to make the implementation closer to the formal system, we can easily simulate right-appending by introducing a new operator `(<+)` which is basically just list-cons with the operands flipped. Finally, local contexts are monoids, since they are just lists.

There are many utility functions that work on local contexts to split, filter and query them, but we shall not show them in this report for space reasons.

7.3.2 Global contexts

Global contexts are modeled by four separate maps as seen in Listing 14.

```haskell
newtype FreeCtx a     = FreeCtx     { unFreeCtx     :: M.Map Name (PolyType a) }
newtype KindCtx a     = KindCtx     { unKindCtx     :: M.Map Name Kind }
newtype DestrCtx a    = DestrCtx    { unDestrCtx    :: M.Map Name (Destr a) }
newtype InstanceCtx a = InstanceCtx { unInstanceCtx :: M.Map Name [ClassInstance a] }
data ClassInstance a = ClassInstance
  { ciClassName        :: Name
  , ciInstanceTypeName :: Name
  , ciParams           :: [Name]
  , ciDictionary       :: M.Map Name (PolyType a, Expr a)
  }
data Destr a = Destr
  { name   :: Name
  , typ    :: PolyType a
  , bound  :: [(Name, Kind)]
  , args   :: [PolyType a]
  }
```

Listing 14: The global context consists of four separate maps.

Splitting the global context up in this way allows for both better performance and a natural segregation between the namespaces of the different concepts. `FreeCtx` maps top-level names to their types, which includes constructors for data-types. `KindCtx` maps types to their kinds (as in $\mathcal{T} : \chi$). `DestrCtx` takes care of pattern matches. We have adopted a new nomenclature for patterns, calling them "Destructors" to not clash with syntactical patterns. Destructors split up the types of their corresponding constructors, so that we can quickly lookup their structure, which mirrors their syntax in the algorithmic system $\mathcal{P} : \forall \vec{\hat{\alpha}}.\ \hat{B}_1 \to \cdots \to \hat{B}_n \to \hat{\tau}$. Finally `InstanceCtx` handles our simplified pseudo-typeclass system consisting only of derived functors. The data-type for class-instances allow for more type-classes than functors, for future extensions of the implementation.

7.3.3 The typing monad

The typing monad is the monad in which all computations that model type-inference and type-checking are described. The monad is a stack of so-called "monad transformers", that allows us to quite easily describe the type of effects our computations can have. Listing 15 shows the Haskell code for the monad.

```
newtype TypingM a r =
  Typ { unTypingM :: ExceptT (TypingErr a)
                             (RWS (TypingRead a) (TypingWrite a) TypingState)
                             r
      }
data TypingRead a  =
  TR { trCtx       :: LocalCtx a
     , trFree      :: FreeCtx a
     , trKinds     :: KindCtx a
     , trDestrs    :: DestrCtx a
     , trInstances :: InstanceCtx a
     }
data TypingState = TS { names :: Integer -- An integer for generating names
                      , level :: Integer -- For debugging
                      }
type TypingWrite a = [(Integer, Doc ())]
type TypingErr a   = (TyExcept a, LocalCtx a)

runTypingM :: TypingM a r -> TypingRead a -> TypingState
           -> (Either (TypingErr a) r, TypingState, TypingWrite a)
runTypingM tm r s = runRWS (runExceptT (unTypingM tm)) r s
```

Listing 15: The typing monad.

The `TypingM` monad is parameterized over the type of its annotations `a` and its result-type `r`. The monad-stack has `ExceptT` as the outer monad, which is a bit non-standard. It is more common to have `ExceptT` on the inner-most level of the stack, to signify a computation that either succeeds or fails. In this case, having `ExceptT` on the outer level allows us to still get information about the state of the computation when it failed. This can be seen in the type signature of of `runTypingM` which returns a triple of the perhaps-failed result, the final state of the computation, and the "log" it produced. This is connected with the `RWS` monad which stands for "Reader-Writer-State" monad. A `Reader` monad simply represents a computation that can read some information. Conversely, a `Writer` monad can only *write* some information. Finally, a `State` monad can both read and write. The `RWS` monad thus allows us to specify at the type-level what things a computation can read, write, and access statefully. It is important to note that all these effects are not "true" effects in the sense that there is no actual mutation going, and everything is "simulated" using pure computations.

The `TypingRead` data-type models the entire typing context, with all the maps and the local context aggregated into one. The state is most importantly an integer that is used to generate new fresh names using the `MName` constructor previously discussed. The `TypingWrite` type allows a typing computation to emit its progress with regards to the algorithmic typing rules, which is very useful for debugging and error messages. Finally, when an error happens, there is an exception (which is a rather big enum-style data-type not shown here) and the typing-context in which the exception happened is reported.

7.3.4 The typing judgments

The typing judgments formulated in section 5.3 are readily implementable in Haskell using the data-types we have presented so far. There is a Haskell function that corresponds to each judgment, as seen in Listing 16, although patterns have been split into two functions.

```
check      :: Expr a -> PolyType a -> TypingM a (LocalCtx a)             -- Γ ⊢ e ⇐ A ⊣ Δ
synthesize :: Expr a -> TypingM a (PolyType a, LocalCtx a)               -- Γ ⊢ e ⇒ A ⊣ Δ
applysynth :: PolyType a -> Expr a -> TypingM a (PolyType a, LocalCtx a) -- Γ ⊢ A • e ⇒⇒ C ⊣ Δ
instL :: Name -> PolyType a -> TypingM a (LocalCtx a) -- Γ ⊢ α^ : χ :<= A ⊣ Δ
instR :: PolyType a -> Name -> TypingM a (LocalCtx a) -- Γ ⊢ A =<: α^ : χ ⊣ Δ
subtypeOf  :: PolyType a -> PolyType a -> TypingM a (LocalCtx a)         -- Γ ⊢ A <: B ⊣ Δ
checkPat   :: Pat a -> PolyType a -> TypingM a (LocalCtx a)              -- Γ ⊢ p ↗↙ A ⊢ Δ
checkDestr :: Destr a -> [Pat a] -> PolyType a -> TypingM a (LocalCtx a) -- Γ ⊢ (C p̄) ↗↙ A ⊢ Δ
```

Listing 16: The Haskell signatures corresponding to the typing judgments.

The type signatures are in general completely mechanically derived from the form of the judgments. Note that the input context is not present in the types because it is a field of the `TypingM` monad's `Reader` type.

There are also Haskell equivalents of the kind-judgment and context well-formedness, but we shall not discuss them here for brevity.

7.3.5 Type-checking

Type-checking is implemented by the `check` function. The definition is rather large so we shall not reproduce it here in its entirety, but Listing 17 shows the overal structure of its definition. The first case is shown un-redacted. The rest of the cases are either totally omitted as indicated by `...`, or somewhat simplified by removing a few lines that are not important to understanding the core behaviour.

```haskell
check :: Expr a -> PolyType a -> TypingM a (LocalCtx a) -- Γ ⊢ e ⇐ A ⊣ Δ
check e@(A eann e') ty@(A tann ty') = check' e' ty' where
  -- ∀I
  check' _ (Forall alpha k aty) = do
    rule "∀I" (pretty e <+> "<=" <+> pretty ty)
    alpha' <- freshName
    let alphat = A tann $ TVar alpha'
    let aty' = subst alphat alpha aty
    let esubst = substTVarInExpr alphat alpha e
    outctx <- withCtx (\g -> g <+ Uni alpha' k) (branch $ check esubst aty')
    (delta, _, _) <- splitCtx (Uni alpha' k) outctx
    pure delta

  -- ->I
  check' (Lam x mty e2) (aty :->: bty) = do
    ctx' <- maybe getCtx (aty `subtypeOf`) mty
    let c = (LamB, x) `HasType` aty
    (delta, _, _) <- splitCtx c =<< withCtx (const $ ctx' <+ c) (branch $ check e2 bty)
    pure delta

  -- Case<=
  check' (Case on clauses) _ = ...

  -- TickAbsI
  check' (TickAbs af k e2) (Later k' t2) = ...

  -- Tuple<=
  check' (Tuple es) (TTuple ts) = ...

  -- Let<=
  check' (Let p e1 e2) _ = do
    (ty1, ctx') <- branch $ synthesize e1
    ty1s <- substCtx ctx' ty1 `decorateErr` (Other "[Let<=]")
    case p of
      A _ (Bind nm) -> withCtx (const $ ctx' <+ ((LetB, nm) `HasType` ty1s)) $ branch $ check e2 ty
      _             -> snd <$> (withCtx (const ctx') $ branch $ checkClause ty1s (p, e2) ty)

  -- Sub
  check' _ _ = do
    (aty, theta) <- branch $ synthesize e
    atysubst <- substCtx theta aty
    btysubst <- substCtx theta ty
    withCtx (const theta) $ branch $ atysubst `subtypeOf` btysubst
```

Listing 17: Redacted version of the `check` function.

`check` is implemented in terms of a helper function `check'`. This is very common in the code, and is a consequence of our choice to use `Annotated` to write our code. Thus, we have to always unwrap a layer of annotation before getting to the actual constructors we wish to pattern match on.

We shall focus on the first case (`∀I`) as that is complete with respect to the source code. The case is executed when the type to check against is polymorphic. The first line `rule...` is some debug-info that logs what rule was triggered. This works together with the `branch` combinator which models that a premise was chosen from the conclusion (in this case `∀I`). The line `alpha' <- freshName` generates a fresh name to be used in place of the universally quantified variable `alpha`. This is a rather significant change compared to the specification, because the algorithmic formulation does not deal with name-capture in any way. But for real programs this is important, and without this step, the programmer essentially

has to come up with globally unique names for type-variables, which is a mess. The following two lines deal with substituting this new fresh name for the old type variable both in the type, but also within the expression `e`. This is because `e` could contain type-annotations that mention the universal variable. Finally, on line 9 the code proceeds into the premise of the rule, by checking the quantified type against the expression `esubst` (which has the fresh variable `alpha'` substituted) in a context with `alpha'` as a universal type-variable. Finally, the output context is split using `splitCtx` and the prefix is returned.

The `∀I` case shows that translating the rules to Haskell is a rather straightforward process, but one must be careful to generate these fresh names and substitute them appropriately. All the other rules follow the same general pattern. The `->I` case shows how we can easily handle both the case where the lambda-bound variable is annotated and when it is not, by using the `maybe` combinator. The bound variable `x` is added to the context with a lambda-binding, and the checking proceeds in this context. The `Let<=` case shows how to check let-bindings. This case models two rules in the specification, namely both Let and LetPat, by matching on the pattern. If the pattern simply is a name, then we bind it and proceed. If not, we continue as if it was a case-of expression with one clause.

Finally, the last case shows subsumption. The `substCtx` function models context substitution, and substitutes away all solved existentials. An important thing to note here is that while the order of inference rules in the specification is unimportant, case-clauses in Haskell are matched top to bottom, so it is vitally important that the subsumption case is put in the bottom. Otherwise, everyhing would match on that case, which is not what we want.

7.3.6 Type-synthesis

Type-synthesis is implemented by the `synthesize` function. It has even more cases than the `check` function! Listing 18 show its definition in a redacted form.

```haskell
synthesize :: Expr a -> TypingM a (PolyType a, LocalCtx a) -- Γ ⊢ e ⇒ A ⊣ Δ
synthesize expr@(A ann expr') = synthesize' expr' where
  -- Var
  synthesize' (Var nm) = do
    ctx <- getCtx
    fctx <- getFCtx
    case (nm `hasTypeInCtx` ctx <|> nm `hasTypeInFCtx` fctx) of
      Just ty -> checkWfType ty *> pure (ty, ctx)
      Nothing -> nameNotFound nm

  -- TickVar
  synthesize' (TickVar nm) = synthesize' (Var nm)
  -- Anno
  synthesize' (Ann e ty) = branch (check e ty) *> ((ty, ) <$> getCtx)
  -- ->I=> (with generalization)
  synthesize' (Lam x Nothing e) = ...
  -- ->AnnoI=>
  synthesize' (Lam x (Just argty) e) = ...
  -- PrimRec=>
  synthesize' (PrimRec prty) = ...
   -- Fmap=>
  synthesize' (Fmap fmapty) = ...

  -- ->E
  synthesize' (e1 `App` e2) = do
    rule "->E" (pretty expr)
    (ty1, theta) <- branch $ synthesize e1
    ty1subst <- substCtx theta ty1
    withCtx (const theta) $ branch $ applysynth ty1subst e2

  -- Prim=>
  synthesize' (Prim p) = ...
  -- Case=>
  synthesize' (Case e clauses) = cannotSynthesize expr
  -- Tuple=>
  synthesize' (Tuple es) = ...
  -- TickAbs=>
  synthesize' (TickAbs nm k e) = ...

  -- TypeApp=>
  synthesize' (TypeApp ex arg) = do
    (exty, theta) <- branch $ synthesize ex
    extys <- substCtx theta exty
    checkWfType arg
    k' <- kindOf arg
    case extys of
      A _ (Forall af k faty)
        | k' == k -> pure (subst arg af faty, theta)
      _           -> otherErr ...

  synthesize' _ = cannotSynthesize expr
```

Listing 18: The redacted implementation of the `synthesize` function.

The `Var` case is worth noting. We do not have separate syntax for local and global names, so we potentially have to lookup the name in both contexts starting with the local one. There are two cases for lambda abstractions; one handles inference when the parameter is not annotated, and the other handles the case

where there is an annotation. Synthesis for application follows the general structure of the corresponding inference rule, and calls the `applysynth` function. The code also shows where our inference totally gives up as case expressions cannot be inferred. In theory, some inference can be recovered by inferring the type of each clause, and then applying subtyping as a least-upper-bound relation to all the inferred types, but even such a prodecure is not guaranteed to find a solution even if it exists, since we do not have principal types.

The case for explicit type application is also somewhat interesting. It first infers the type of the expression in $e\,\{A\}$ and then fully applies the inferred type under the context. Then it checks that the inferred type is well-formed. It then pattern matches on the inferred type to make sure that it is forall-quantified. Importantly, this step will fail if the type is an existential, meaning that this is another limitation of the inference system. The logical next step would be to introduce an existential for the quantification variable `af` and assign it to `arg`, but instead we choose to substitute `arg` directly for `af` in the quantified type `faty`. The first approach would not allow us to have some form of impredicativity introduced by explicit type applications, since the specification relies on all solutions to existentials being mono-types. In fact, the implementation statically disallows such a solution, since the type of the constructor for solved variables only permits monomorphic types as arguments.

7.3.7 Application synthesis

Application synthesis is implemented by the `applysynth` function. Listing 19 shows the redacted code that implements the function.

```haskell
applysynth :: PolyType a -> Expr a -> TypingM a (PolyType a, LocalCtx a) -- Γ ⊢ A • e ⇒⇒ C ⊣ Δ
applysynth ty@(A tann ty') e@(A _ e') = applysynth' ty' e' where
  -- ∀App
  applysynth' (Forall alpha k aty) _ = do
    alpha' <- freshName -- fresh name to avoid clashes
    let atysubst = subst (texists alpha') alpha aty
    withCtx (\g -> g <+ Exists alpha' k) $ branch $ applysynth atysubst e

  -- |>kαApp
  applysynth' (TExists alpha) (TickVar _tv) =
    let articulate a1 a2 =
          let articulated = A tann $ Later (texists a1) (texists a2)
          in  [Exists a2 Star, Exists a1 ClockK, alpha := articulated]
    in  appsynthexists alpha "|>kαApp" articulate

  -- αApp
  applysynth' (TExists alpha) _ =
    let articulate a1 a2 =
          let articulated = A tann (texists a1 :->: texists a2)
          in  [Exists a2 Star, Exists a1 Star, alpha := articulated]
    in  appsynthexists alpha "αApp" articulate

  -- |>kApp◇
  applysynth' (Later kappat cty) (Prim Tick) = do
    kappa <- extractKappa kappat
    ctx <- getCtx
    ctx `mustBeStableUnder` kappa
    pure (cty, ctx)

  -- |>kApp
  applysynth' (Later kappat cty) _ = (cty,) <$> branch (check e kappat)

  -- ->App
  applysynth' (aty :->: cty) _ = (cty,) <$> branch (check e aty)

  applysynth' _ _ = cannotAppSynthesize ty e

  appsynthexists alpha ruleName toInsert = do
    ctx <- getCtx
    if ctx `containsEVar` alpha
      then do
        a1 <- freshName; a2 <- freshName
        ctx' <- insertAt (Exists alpha Star) (toInsert a1 a2)
        delta <- branch $ withCtx (const ctx') $ check e (texists a1)
        pure (texists a2, delta)
      else nameNotFound alpha
```

Listing 19: The application-synthesis function.

The first case implements the ∀App rule, which introduces a fresh name into the context when an expression `e` is applied to a forall-quantified type. The next case implements application. There are two cases here, depending on if the expression is a "normal" expression or a tick-variable. The two cases are quite similar, and therefore both implemented with a helper function `appsynthexists`. They essentially both follow their algorithmic specification, and articulate the existential variable to have the shape of either a function arrow or a later-type. `|>App◇` implements the rule where the tick-constant is applied to a term of a later-type. In this case, we must check if the context is stable under the clock-variable, which the `mustBeStableUnder` function does. Note that we cannot type-check applying the tick-constant

to a term of an existential type, since we cannot know what clock-variable the context must be stable under. This is fixable, by deferring the check to a time where the clock-variable is solved, but we have not implemented it here as it is rarely a problem.

The final two cases are simple one-liners. `(cty,) <$> e` for some monadic expression `e` will take the result of `e` and put it as the second element in a tuple where the first is `cty`[3]. The final catch-all case simply spits out an error signifying that we could not apply-synthesize a type for the arguments.

7.3.8 Subtyping

Subtyping is implemented by the `subtypeOf` function. It is rather large (over 100 lines), but follows the algorithmic specification of the subtyping relation very closely. Listing 20 shows the redacted implementation of `subtypeOf`.

[3]Using a GHC extension called `TupleSections`.

```haskell
subtypeOf  :: PolyType a -> PolyType a -> TypingM a (LocalCtx a) -- Γ ⊢ A <: B ⊣ Δ
subtypeOf ty1@(A ann1 typ1) ty2@(A ann2 typ2) = subtypeOf' typ1 typ2 where
  -- <:Free
  subtypeOf' (TFree x) (TFree x') = ...

  -- <:Var
  subtypeOf' (TVar x) (TVar x') = ...

  -- <:Exvar
  subtypeOf' (TExists a) (TExists a') = ...

  -- <:->
  subtypeOf' (a1 :->: a2) (b1 :->: b2) = do
    theta <- branch (b1 `subtypeOf` a1)
    a2' <- substCtx theta a2
    b2' <- substCtx theta b2
    withCtx (const theta)  -- <:∀R

  -- <:∀R
  subtypeOf' _t1 (Forall alpha k (A ann t2)) = do
    alpha' <- freshName
    let ty2' = subst (A ann $ TVar alpha') alpha (A ann t2)
    theta <- withCtx (\g -> g <+ Uni alpha' k) $ branch (ty1 `subtypeOf` ty2')
    pure $ dropTil (Uni alpha' k) theta

  -- <:∀L
  subtypeOf' (Forall alpha k (A at1 t1)) _ = do
    alpha' <- freshName
    let t1' = subst (A at1 $ TExists alpha') alpha (A at1 t1)
    theta <- withCtx (\g -> g <+ marker alpha' <+ Exists alpha' k) $ branch (t1' `subtypeOf` ty2)
    pure $ dropTil (Marker alpha') theta $ branch (a2' `subtypeOf` b2')

  -- <:TApp
  subtypeOf' (TApp a1 a2) (TApp b1 b2) = ...

  -- <:Rec
  subtypeOf' (RecTy b1) (RecTy b2) = branch (b1 `subtypeOf` b2)

  -- <:Tuple
  subtypeOf' (TTuple ts1) (TTuple ts2) = ...

  -- <:Later
  subtypeOf' (Later a1 a2) (Later b1 b2) = ...

  -- <:InstantiateL
  subtypeOf' (TExists ahat) _
    | ahat `inFreeVars` ty2 = occursIn ahat ty2
    | otherwise = checkWfType (A ann1 $ TExists ahat) *> branch (ahat `instL` ty2)

  -- <:InstantiateR
  subtypeOf' _ (TExists ahat) = ...

  subtypeOf' t1 t2 = cannotSubtype ty1 ty2
```

Listing 20: The implementation of the subtyping relation.

The first case whose implementation is shown deals with subtyping function types. As in the specification,

function types are contravariant, which is why the first line in the do-block calls `b1 `subtypeOf` a1`. In the next premise (the last line of that case), the types are fully applied with the intermediate context `theta` using the `substCtx` function.

The next two cases (`<:∀R` and `<:∀L`) again show that we have to supply fresh names every time we introduce a quantifier, in order to avoid name-clashes. They also both show the use of the `dropTil` function which drops context-elements to the right of its first argument, thus implementing the "pattern-matching" syntax on output contexts in the implementation.

Finally, the `<:InstantiateL` case shows the use of Haskell guards to check for cycles during subtyping. It models the traditional occurs-check also known from standard Hindley-Damas-Milner inference. If no cycles are detected, it switches to the left-instantiation function `instL`. The `<:InstantiateR` case is not shown here but is fully symmetric to its left counterpart.

7.3.9 Instantiation

Type instantiation is implemented with two functions: `instL` and `instR` corresponding to left-instantiation and right-instantiaton. The two functions are very similar, so we shall only show some of the implementation of one of them, namely `instL` seen in Listing 21.

```haskell
instL :: Name -> PolyType a -> TypingM a (LocalCtx a) -- Γ ⊢ α^ : χ :<= A ⊣ Δ
instL ahat ty@(A a ty') =
  instLSolve ahat ty `catchError` \err ->
    case ty' of
      -- InstLReach
      TExists bhat -> do
        ak <- queryKind ahat; bk <- queryKind bhat
        Exists ahat ak `before` Exists bhat bk >>= \case
          True -> assign bhat (A a $ TExists ahat)
          False -> otherErr $ "[InstLReach] error"

      -- InstLArr
      t1 :->: t2 -> do
        af1 <- freshName; af2 <- freshName
        let ahat1 = Exists af1 Star
            ahat2 = Exists af2 Star
            arr   = A a $ A a (TExists af1) :->: A a (TExists af2)
        ctx' <- insertAt (Exists ahat Star) [ahat2, ahat1, ahat := arr]
        theta <- withCtx (const ctx') $ branch (t1 `instR` af1)
        substed <- substCtx theta t2
        withCtx (const theta) $ branch (af2 `instL` substed)

      -- InstLAllR
      Forall beta k bty -> do
        beta' <- freshName
        let bty' = subst (A a $ TVar beta') beta bty
        theta <- withCtx (\g -> g <+ Uni beta' k) $ branch (ahat `instL` bty')
        pure $ dropTil (Uni beta' k) theta

      -- InstLTApp. Nearly identical to InstLArr
      TApp t1 t2 -> ...

      -- InstLTuple
      TTuple ts -> ...

      -- InstLLater. Similar to instantiation of other type-combinators
      Later t1 t2 -> ..

      -- InstLRec
      RecTy t -> ...

      _ -> throwError err
  where
    instLSolve ahat ty = do
      mty <- asMonotypeTM ty
      assign ahat mty
```

Listing 21: Implementation of left-instantiation.

When instantiating any existential, we first try to see if the type we are attempting to assign is a monotype. If it is, we simply attempt to assign the variable the type. This will fail many times, because the `assign` function will *always* check if the context is valid after assignment. Consequently, if the type-to-be-assigned `ty` contains any existentials that are to the right of the variable `ahat` in the local context, the resulting context will not be well-formed. Since `instLSolve` failed, the remaining cases will be tried thanks to the `catchError` combinator. Note that these errors are completely purely modeled by the `Either` monad behind the scenes. Thus, they are *not* classical runtime exceptions.

The rest of the function corresponds pretty closely with the specification. `InstLReach` swaps the assignment of one existential to another if their positions in the context make it possible. `InstLArr` generates two new fresh existential variables and inserts them in front of the assignee's position in the context before proceeding contravariantly. `InstLAllR` also follows its specification closely; it generates a fresh universal type-variable `beta'`, substitutes it for `beta` and adds it to the context before continuing to assign `ahat` to the quantified type. The last line maintains the scope by dropping the context from the right until it encounters `beta`. Note that `dropTil` is inclusive and so `beta` is also absent from the output context.

7.3.10 Pattern matching

Pattern matching is implemented by two functions: `checkPat` and `checkDestr`, both shown in Listing 22.

```
1  -- check that patterns type-check and return a new ctx extended with bound variables
2  checkPat :: Pat a -> PolyType a -> TypingM a (LocalCtx a)
3  checkPat pat@(A ann p) ty = do
4    ctx <- getCtx -- get local context
5    dctx <- getDCtx -- get destructor context
6    case p of
7      Bind nm -> pure $ ctx <+ (LetB, nm) `HasType` ty
8
9      Match nm pats -> case query nm dctx of
10       Nothing    -> otherErr $ "Pattern " ++ show nm ++ " not found in context."
11       Just destr -> branch $ checkDestr destr pats ty
12
13     PTuple pats ->
14       -- generate a tuple destructor of length (length pats)
15       let plen   = genericLength pats
16           dname  = UName $ "tuple_" ++ show plen
17           dbound = map (\x -> (DeBruijn x, Star)) [0 .. (plen - 1)]
18           dargs  = map (A ann . TVar . fst) dbound
19           dtyp   = A ann (TTuple dargs)
20           d      = Destr {name = dname, bound = dbound, typ = dtyp, args = dargs}
21       in  branch (checkDestr d pats ty)
22
23 -- in a context, check a destructor against a list of patterns and an expected type.
24 -- if it succeeds, it binds the names listed in the pattern match to the input context
25 checkDestr :: Destr a -> [Pat a] -> PolyType a -> TypingM a (LocalCtx a)
26 checkDestr d@(Destr {name, args}) pats expected@(A ann _)
27   | length pats /= length args = otherErr ...
28   | otherwise                  = do
29       (delta, Destr {typ = etyp, args = eargs}) <- existentialize ann d
30       ctx' <- withCtx (const delta) $ branch $ etyp `subtypeOf` expected
31       foldlM folder ctx' $ zip pats eargs
32       where
33         folder acc (p, t) = do
34           t' <- substCtx acc t
35           withCtx (const acc) $ checkPat p t'
```

Listing 22: Pattern matching in CloFRP.

`checkPat` checks a pattern against a type, returning an output context with pattern-bound names (if successful). It matches on the AST of the pattern; if it is just a `Bind` pattern (x) then it binds the name to the context and returns. If it is a constructor-pattern, it queries the destructor in the context and then delegates its work to `checkDestr`. Finally, tuple-patterns are cleverly implemented by generating a destructor for whatever length of tuple we are expecting on the fly, and then delegating to `checkDestr` again.

`checkDestr` also follows its specification quite closely, with one exception. The destructor-context in the specification stores destructors whose type-parameters are already existentials. In the implementation, the types in the destructor-context are not existential types, and thus need to be `existentialized` on the fly. This is because we need to generate fresh names anyway, so there is no point in doing this in a pre-processing step. The `foldlM` function implementes the left-iteration over the sub-pattern types seen in the specification. The post-fix `M` indicates that it is `foldl` but in a monadic context. Thus its goes through every subpattern left-to-right and checks it against the arguments of the existentialized destructor.

That concludes the brief walkthrough of the type-inference and type-checking implementation.

7.4 Type-synonyms

Program elaboration is the act of transforming the parsed declarations into a form that is suitable for type-checking. The most complicated part of this is generating derived code (Section 7.5) and expanding type-synonyms. Recall how type-synonyms are represented in our implementation of CloFRP:

```
data Synonym a =
  Synonym
    { synName      :: Name
    , synBound     :: [(Name, Kind)]
    , synExpansion :: PolyType a
    }
```

A type-synonym has a name, a list of bound type-parameters and an expansion. For example, the type-synonym `type Pair a b = (a,b)` would be roughly represented as

```
Synonym
  { synName      = "Pair"
  , synBound     = [("a", Star), ("b", Star)]
  , synExpansion = TTuple ["a", "b"]
  }
```

Type-synonyms are in many ways type-level functions, but with extra restrictions: You cannot partially apply a type-synonym, and type-synonyms cannot be recursive in any way.

Checking that a type-synonym is not recursive can be accomplished relatively simply, as shown in Listing 23.

```
checkRecSyn :: Name -> PolyType a -> TypingM a ()
checkRecSyn name (A _ ty') =
  case ty' of
    TFree n
      | n == name                    -> otherErr $ show name ++ " is recursive"
      | Just syn' <- M.lookup n syns -> checkRecSyn name (synExpansion syn')
      | otherwise                    -> pure ()

    TVar _          -> pure ()
    TExists _       -> pure ()
    TApp t1 t2      -> checkRecSyn name t1 *> checkRecSyn name t2
    t1 :->: t2      -> checkRecSyn name t1 *> checkRecSyn name t2
    Forall _n _k t -> checkRecSyn name t
    RecTy  t        -> checkRecSyn name t
    TTuple ts       -> traverse (checkRecSyn name) ts *> pure ()
    Later _k t      -> checkRecSyn name t
```

Listing 23: Checking whether a synonym is recursive.

`checkRecSyn` is applied to the body (expansion) of each type-synonym. If a type-synonym is directly recursive, then it will mention itself in its body, and the case on line 5 will be hit. When it encounters another synonym `syn'`, it must check whether the body of the synonym mentions itself, to check for mutual recursion. For example, `type Foo = Bar. type Bar = Foo` is not permitted. The rest of the cases just recurse down the AST of types.

Expanding type-synonyms in the source is not entirely straightforward. In many ways, it is similar to β-reduction. The problem is we do not have a constructor in the AST of types to describe type-level functions, since they are not really valid types. Instead, we augment the AST of types with a special-purpose data-type:

```
1  data SynonymExpansion a
2    -- a fully expanded synonym
3    = Done (PolyType a)
4    -- a synonym that still needs at least one application
5    | Ex Name (PolyType a -> SynonymExpansion a)
```

We can then convert from the `Synonym` data-type to this representation as follows:

```
1  synonymExpansion :: a -> Synonym a -> SynonymExpansion a
2  synonymExpansion ann = go 0 . deb where
3    deb syn@(Synonym { synBound = b, synExpansion = ex }) =
4      syn { synExpansion = deBruijnify ann (map fst b) ex }
5
6    go i syn@(Synonym { synName = nm, synBound = b, synExpansion = ex }) =
7      case b of
8        [] -> Done ex
9        _:xs -> Ex nm $ \t ->
10           let ex' = subst t (DeBruijn i) ex
11           in  go (i+1) (syn { synBound = xs, synExpansion = ex' })
```

The above definition first "deBruijnifies" the type-synonym by replacing the bound variables in its expansion with their deBruijn indices. This is to avoid name-capture problems. It then recurses over the bound variables – if there are none, then the expansion is finished. If there is at least one bound variable, then it returns a function that, when given a type, will substitute that type in for the bound variable.

Listing 24 shows the code that traverses a type and recursively expands all alisases in it. The `go` helper function traverses the type – if it encounters a free type, it will look up if it is a type-synonym (since there is no syntactical way to distinguish between type-synonyms and normal types), and then return its `SynonymExpansion` representation. The most interesting recursive case is the case for type-application `TApp`. The case recurses down on both arguments to `TApp` and pattern-matches on their results, using the `>>*=` operator which is just monadic `bind` on tuples. If neither result is a partially-applied type-synonym, then the result is just normal type-application. However, if the first argument only results in a partially-applied type-synonym (the `Ex` constructor), then the synonym is expanded with the result of the second argument (`t2'`). In any other case, an error is raised by `wrong`.

Finally, the `expandSynonym` function actually runs `go` in a fixpoint, in order to expand synonyms deeply. It will keep expanding until one expansion is equal to the input type. The `=%=` operator models this by comparing two types modulo their annotations (such as a source-code positions).

```
expandSynonyms :: forall a. Synonyms a -> PolyType a -> TypingM a (PolyType a)
expandSynonyms syns t = go t >>= \case
    Done t' | t =%= t' -> pure t'
            | otherwise -> expandSynonyms syns t'
    Ex nm _ -> wrong nm
  where
    go :: PolyType a -> TypingM a (SynonymExpansion a)
    go (A ann ty') =
      case ty' of
        TFree n
          | Just syn <- M.lookup n syns -> pure $ synonymExpansion ann syn
          | otherwise                   -> done (A ann $ ty')

        TVar _     -> done (A ann ty')
        TExists _  -> done (A ann ty')
        TApp t1 t2 -> (go t1, go t2) >>*= \case
            (Done t1', Done t2') -> done (A ann $ TApp t1' t2')
            (Done _, Ex nm _)    -> wrong nm
            (Ex _ f1, Done t2')  -> pure $ f1 t2'
            (Ex nm _, Ex _ _)    -> wrong nm

        t1 :->: t2 -> (go t1, go t2) >>*= \case
            (Done t1', Done t2') -> done (A ann $ t1' :->: t2')
            (Ex nm _, _)         -> wrong nm
            (_, Ex nm _)         -> wrong nm

        Forall n k t1 -> go t1 >>= \case
            Done t1' -> done $ A ann $ Forall n k t1'
            Ex nm _ -> wrong nm

        RecTy t1 -> go t1 >>= \case
            Done t1' -> done $ A ann $ RecTy t1'
            Ex nm _ -> wrong nm

        TTuple ts -> done . A ann . TTuple =<< traverse fn ts where
          fn tt = go tt >>= \case
            Done tt' -> pure tt'
            Ex nm _ -> wrong nm

        Later k t1 ->
          go t1 >>= \case
            Done t1' -> done $ A ann $ Later k t1'
            Ex nm _ -> wrong nm
```

Listing 24: Expanding all synonyms in a type.

7.5 Deriving functors

As mentioned before, CloFRP supports deriving functor instances. It does not however feature a type-class system otherwise. Functorial mappings are only needed to implement primitive recursion for strictly positive type-constructors, since the semantics of $\mathsf{primRec}_F$ is defined in terms of fmap_F. Thus, to be frank, the entire thing is kind of hackish, but it works well enough to serve its purpose. Deriving `fmap` for a type is quite abstract, so let us start with an example. Consider the below data-declaration:

```
data T a f = C1 (a -> f) | C2 Int (Maybe f)
```

If we would write a functor instance for this in Haskell by hand it would look like this:

```haskell
instance Functor (T a) where
  fmap f t =
    case t of
      C1 g    -> C1 (\x -> f (g x))
      C2 i mg -> C2 i (fmap f mg)
```

We can write code to automatically derive such implementations based on the constructors of a type. However, the generated code is not quite as elegant. In the end, the derived code for the above declaration would look like this:

```haskell
instance Functor (T a) where
  fmap f t =
    case t of
      C1 x0    -> C1 ((\x b -> f (x (id b))) x0)   -- β-reduces to C1 (\b -> f (x0 b))
      C2 x0 x1 -> C2 (id x0) ((\x -> fmap f x) x1) -- β-reduces to C2 x0 (fmap f x1)
```

As one can see, there is an excess of functions and function-applications, but it β-reduces to an α-equivalent definition to what we originally wrote. The scheme of applying a lambda to every argument of the constructors make the implementation slightly simpler, as one can more easily avoid naming conflicts.

Listing 25 shows the code that implements functor-deriving.

```haskell
deriveFunctor :: Datatype a -> Either String (ClassInstance a)
deriveFunctor (Datatype {dtName, dtBound = []}) =
  Left $ show $ "Cannot derive functor for concrete type" <+> pretty dtName
deriveFunctor (Datatype {dtName, dtConstrs = []}) =
  Left $ show $ "Cannot derive functor for uninhabited type" <+> pretty dtName
deriveFunctor (Datatype {dtName, dtBound = bs@(b:_), dtConstrs = cs@(A ann _ : _)}) = do
  let (bnm, bk) = safeLast b bs -- the last bound variable in bs, or b if bs = []
  checkKind (bnm, bk) -- check that the kind of bnm is correct (*)
  let ?annotation = ann -- use ann for implicit anotations
  expr <- deriveFmapDef bnm cs -- derive the definition of the functor
  let extrabs = map fst $ safeInit bs -- take all variables except the last
  let nfa = foldl' tapp (tfree dtName) (map tvar extrabs) -- The nearly fully applied type
  -- the type for fmap
  let typ = forAllK (safeInit bs ++ [(bnm, Star), ("#b", Star)]) $
              (tvar bnm `arr` tvar "#b") `arr` (nfa `tapp` tvar bnm) `arr` (nfa `tapp` tvar "#b")
  -- create the class instance
  let inst = ClassInstance { ciClassName        = "Functor"
                           , ciInstanceTypeName = dtName
                           , ciParams           = extrabs
                           , ciDictionary       = M.singleton "fmap" (typ, expr)
                           }
  pure $ inst

deriveFmapDef :: (?annotation :: a) => Name -> [Constr a] -> Either String (Expr a)
deriveFmapDef tnm cs =
  let vn = UName "val"
      fn = UName "f"
  in  lam' fn . lam' vn .
        casee (var vn) <$> traverse (deriveFmapConstr (var fn) tnm) cs
```

Listing 25: The implementation of `deriveFunctor` and `deriveFmapDef`.

The first two cases simply fail when we certainly know that we cannot derive `fmap` for the type. The last case sets up the necessary information to actually derive `fmap` in `deriveFmapDef` function. It also creates the type of `fmap` which is `(a -> #b) -> f a -> f #b` where `f` is a functor. The hash in `"#b"` is a trick to make sure that it is a fresh type-variable – hashes are not legal syntactic names, so the user cannot already have used it. For example, if the functor is `ListF a f`, then the last parameter `f` is the functorial parameter, so the generated type of `fmap` for `ListF a f` is `forall a f #b. (f -> #b) -> ListF a f -> ListF a #b`.

`deriveFmapDef` generates the body of fmap, introducing two lambdas to bind the first two arguments. It then creates a pattern-matching clause for each constructor in the data-type, using `deriveFmapConstr`. We shall not show the code for `deriveFmapConstr` here, for brevity, but it simply shallowly matches on the constructor, binding a name to each argument. The interesting bit then happens when we derive a function to be applied to each bound name based on the type of its associated argument in `deriveFmapArg` (Listing 26).

```
deriveFmapArg :: Expr a -> TVarName -> PolyType a -> Either String (Expr a)
deriveFmapArg f tnm typ@(A anno _) = go typ where
  go typ2 | not (inFreeVars tnm typ2) = pure ide
  go (A ann typ') =
    let ?annotation = ann
    in  case typ' of
      TFree _    -> pure ide
      TExists _ -> Left "existentials are not part of the concrete syntax"
      TVar nm | nm == tnm -> pure $ f
              | otherwise -> pure ide
      Forall v k t -> go t

      Later t1 t2 -> do
        let af = UName "af"
        e2 <- go t2
        pure $ lam' "x" $ tAbs af k (e2 `app` (var "x" `app` tickvar af))

      TTuple ts    -> deriveFmapTuple f tnm ts
      t1 `TApp` t2 -> pure $ lam' "x" $ (A anno $ Fmap t1) `app` f `app` var "x"

      t1 :->: t2 -> do
        e1 <- cogo t1
        e2 <- go t2
        let (x,b) = (UName "x", UName "b")
        pure $ lam' x $ lam' b $ e2 `app` (var x `app` (e1 `app` var b))

      RecTy t -> Left $ "Cannot derive functor for recursive types"

  cogo typ2 | not (inFreeVars tnm typ2) = pure ide
  cogo (A ann typ') =
    let ?annotation = ann
    in  case typ' of
      -- identical to go except
      TVar nm | nm == tnm -> Left $ "type variable " ++ pps tnm ++ " is in a negative position"
              | otherwise -> pure ide

      t1 :->: t2 -> do
        e1 <- go t1
        e2 <- cogo t2
        let (x, b) = (UName "x", UName "b")
        pure $ lam' x $ lam' b $ e2 `app` (var x `app` (e1 `app` var b))

  ide = A anno $ Lam "x" Nothing (A anno $ Var "x")
```

Listing 26: The implementation of `deriveFmapArg`.

`deriveFmapArg` checks the type of each argument to a constructor. The `?annotation` binding binds an implicit parameter – this allows us to construct expressions more readably by forgetting about the annotation when using helper functions like `lam', tAbs` or `var`. If the functorial variable `tnm` is not free in the type, then it simply returns the identity function, since the argument is then not part of the functorial structure. If it is, then it pattern matches on the type of the argument. If the argument is exactly `tnm` then the mapping `f` is returned. Most of the other cases simply recurse down the types and combine with the appropriate combinator. The interesting case is for the arrow type. Here, we switch to the `cogo` helper function, which is almost identical to `go` except that it fails if it encounters the functorial type-variable `tnm`. It will also switch back to `go` if it in turn encounters a function type. This dual definition makes sure that we fail on any types that are not strictly positive in `tnm`. As such, arguments of type `a -> tnm` and `(tnm -> a) -> a` are both accepted, but `tnm -> a` is rejected.

The generated code is type-checked along with the rest of the program, so we should be assured that any generated code is safe.

That concludes the very brief rundown of functor deriving. It is a slightly hairy subject, but not of prime importance to this project so it will not be devoted any additional attention.

7.6 Evaluation

This section briefly describes the intepreter that evaluates CloFRP programs. The interpreter is more-or-less a straightforward implementation of the operational semantics.

The data-types used for implementing evaluation can be seen in listing 27.

```
1  -- |A Value is an expression that is evaluated to normal form
2  data Value a
3    = Prim PrimVal
4    | Var Name
5    | TickVar Name
6    | Closure (Env a) Name (Expr a)
7    | TickClosure (Env a) Name (Expr a)
8    | Tuple [Value a]
9    | Constr Name [Value a]
10   | Fold (Value a)
11
12 data PrimVal
13   = IntVal Integer
14   | Tick
15   | RuntimeErr String
16   | FoldP
17   | UnfoldP
18
19 type Globals a = Map Name (Value a)
20 newtype Env a = Env { unEnv :: Map Name (Value a) }
21
22 data EvalRead a =
23   EvalRead { erEnv :: Env a
24            , erGlobals :: Globals a
25            , erInstances :: InstanceCtx a
26            , erInputs :: Inputs a
27            }
28
29 newtype EvalM a r = Eval { unEvalM :: Reader (EvalRead a) r }
```

Listing 27: Data-types for evaluation.

Values correspond very closely to the syntax given in Section 6.1. There are some extra things to handle built-in integers, runtime errors and applications of fold and unfold (since they can be partially applied in CloFRP). The evaluation monad `EvalM` is simply a `Reader` monad, which reads a value of type `EvalRead`. `EvalRead` contains a local environment `Env`, global definitions `Globals`, a map of functor instances `InstanceCtx` and some inputs `Inputs`. Inputs are only really important when receiving values from Haskell, and we shall cover interoperation with Haskell in a later section.

The step-semantics is implemented by the `evalExprStep` function, show in Listing 28.

```haskell
evalExprStep :: Pretty a => Expr a -> EvalM a (Value a)
evalExprStep (A ann expr') =
  let ?annotation = ann
  in  case expr' of
    E.Prim p         -> evalPrim p
    E.TickVar nm     -> pure $ TickVar nm
    E.Lam x _mty e   -> (\env -> Closure env x e) <$> getEnv
    E.TickAbs x _k e -> (\env -> TickClosure env x e) <$> getEnv
    E.Fmap t         -> fmapFromType t
    E.PrimRec t      -> evalPrimRec t
    E.Ann e _t       -> evalExprStep e
    E.TypeApp e _t   -> evalExprStep e
    E.Tuple ts       -> Tuple <$> sequence (map evalExprStep ts)
    E.Case e1 cs     -> evalExprStep e1 >>= (\v1 -> evalClauses v1 cs)

    E.Var nm
      | isConstructor nm -> pure $ Constr (E.UName nm) []
      | otherwise        -> lookupVar nm

    E.App e1 e2 -> do
      v1 <- evalExprStep e1
      v2 <- evalExprStep e2
      case (v1, v2) of
        (Closure cenv nm e1', _) -> do
          let cenv' = extendEnv nm v2 cenv
          withEnv (const cenv') $ evalExprStep e1'

        (TickClosure cenv nm e1', _) -> do
          let cenv' = extendEnv nm v2 cenv
          withEnv (const cenv') $ evalExprStep e1'

        (Constr nm args, _) -> pure $ Constr nm (args ++ [v2])

        (Prim FoldP, _)   -> pure $ Fold v2
        (Prim UnfoldP, _) ->
          case v2 of
            Fold v -> pure v
            _      -> pure . runtimeErr ...

        _ -> pure . runtimeErr ...

    E.Let p e1 e2 -> do
      v1 <- evalExprStep e1
      envE' <- evalPat p v1
      case envE' of
        Right env' -> withEnv (const env') $ evalExprStep e2
        Left err -> pure . runtimeErr $ "Let match failed: " ++ err
```

Listing 28: The redacted implementation of the operational step-semantics.

`evalExprStep` is again a pretty straightforward translation of the operational step-semantics. We have left out a lot of details, such as `evalPat` and evaluating primitives, which includes primitive recursion and fixpoints, but all they do is follow the specification.

The coinductive tick-semantics is implemented by the `evalExprCorec` and `force` functions, seen in Listing 29.

```haskell
evalExprCorec :: Pretty a => Expr a -> EvalM a (Value a)
evalExprCorec expr = go =<< evalExprStep expr where
  go v = do
    case v of
      Constr nm vs -> Constr nm <$> evalMany vs
      Fold v'      -> Fold <$> go v'
      Tuple vs     -> Tuple <$> evalMany vs
      _ -> pure v

  evalMany [] = pure []
  evalMany (v:vs) = do
    v' <- go =<< force v
    (v' :) <$> evalMany vs

force :: Pretty a => Value a -> EvalM a (Value a)
force = \case
  TickClosure cenv nm expr ->
    withEnv (const $ extendEnv nm (Prim Tick) cenv) $ evalExprStep expr
  v -> pure v
```

Listing 29: Implementation of the coinductive tick-semantics.

Once again, it is very close to the semantics specified in Section 6.3. `evalMany` is basically just a monadic right-fold, which is the most natural way to fold over lists in Haskell. This simplistic encoding of the tick-semantics is made possible since Haskell is lazy; as every path of the function only recurses under a constructor, this is essentially using guarded recursion, so the result is a productive evaluation function (given that the programs it evaluates are productive themselves, but that is essentially the whole point of CloFRP).

7.7 Interoperation with Haskell

Since we have implemented CloFRP as a deeply embedded DSL in Haskell, it is pertinent to ask how to integrate CloFRP programs as a component of a Haskell program. We can already get *output* of a CloFRP program using the evaluation functions described in Section 7.6. But this is not quite satisfactory, as the `Value` AST is not especially convenient to work with. There is some clutter, but most importantly the actual data that such a `Value` represents is not reflected in the type-system. Furthermore, we would also like to be able to use Haskell values as *input* to CloFRP programs in a type-safe way.

To accomplish our goal, we will need to somehow reflect the type of a CloFRP program into Haskell's type-system. This is not entirely straightforward, but it can be achieved to some degree with GHC's extensions for type-level programming.

As we saw in Section 7.2.1, with the `DataKinds` extension GHC automatically promotes every suitable data-type to be a *kind*, and its value constructors to be type constructors. A *kind* is the type of a type. For example the kind of `Int` is `*` while the kind of `[]` is `* -> *` and the kind of `Map` is `* -> * -> *`. The only types that are inhabitable have kind `*`.

The key idea is that we want to construct a type based on a value. This would be easy in a dependently typed language, but unfortunately Haskell does not have dependent types. However, with the `DataKinds` extension (along with a few other extensions), we can simulate (limited) dependent types by constructing *singleton types*. A singleton type only has one value (hence the name), and thus a value of a singleton-type has a unique type that represents that value. Therefore, there is an isomorphism between a value of a singleton type and its type, and since types can depend on types in Haskell, we have arrived at dependent types.

In practice, we first create a new data-type to represent CloFRP types that can be reflected, and then a singleton type that lifts it into the type-level (Listing 30).

```haskell
-- | The type of CloFRP-types that can be reflected
data CloTy
  = CTFree Symbol
  | CTTuple [CloTy]
  | CloTy :->: CloTy
  | CloTy :@:  CloTy

-- | Singleton representation to lift CloTy into types
--   using kind-promotion
data Sing :: CloTy -> * where
  SFree  :: KnownSymbol s => Proxy s      -> Sing ('CTFree s)
  SPair  :: Sing t1 -> Sing t2            -> Sing ('CTTuple '[t1, t2])
  STup   :: Sing t  -> Sing ('CTTuple ts) -> Sing ('CTTuple (t ': ts))
  SApp   :: Sing t1 -> Sing t2            -> Sing (t1 ':@:  t2)
  SArr   :: Sing t1 -> Sing t2            -> Sing (t1 ':->: t2)
```

Listing 30: Singleton to lift CloFRP types into the Haskell type-system.

We can see that the first type-parameter of `Sing` has kind `CloTy`. The type-constructors that are prefixed with a single-quote are the promoted data-constructors of `CloTy`. For example, `(':->:)` has kind `CloTy -> CloTy -> CloTy`.

An interesting feature is the `Symbol` type. This is part of Haskell's type-level literals package, which can model natural numbers and strings on the type level. Thus, `Symbol` represents a value that maps to a type-level string. We need this to deal with the open nature of CloFRP's type-system.

The singleton constructor for `CTFree` is `SFree`, which has a `KnownSymbol s` constraint. This tells us that the `Symbol` must be known at compile time. The `Proxy` argument is used to "proxy" the symbol to the value-level. One could call it a value-level witness of a type. Together with `symbolVal :: forall n proxy. KnownSymbol n => proxy n -> String` we can access the type-level string at runtime.

There are two ways two reflect tuple-types: `SPair` is the base-case which reflects a pair, and `STup` extends a tuple to the left. Thus we can represent tuples of any arity (but not below two, which is invalid).

Finally, we can define how to derive a `Sing` that represents an ordinary CloFRP type `PolyType a`. However, we cannot do this at runtime since that would have type `exists (ct :: CloTy). PolyType a -> Sing ct` which is not expressible in Haskell. Luckily for us, we do not actually need this to happen at runtime but at *compile time*. Therefore, we will define a function that uses `TemplateHaskell` to create a singleton from a CloFRP type (Listing 31).

```haskell
-- |Use template haskell to generate a singleton value that represents
-- a CloFRP type
typeToSingExp :: PolyType a -> ExpQ
typeToSingExp (A _ typ') = case typ' of
  P.TFree (P.UName nm) ->
    let nmQ = pure (S.LitT (S.StrTyLit nm))
    in  [| SFree (Proxy :: (Proxy $(nmQ))) |]
  t1 P.:->: t2 ->
    let s1 = typeToSingExp t1
        s2 = typeToSingExp t2
    in  [| $(s1) `SArr` $(s2) |]
  t1 `P.TApp` t2 ->
    let s1 = typeToSingExp t1
        s2 = typeToSingExp t2
    in  [| $(s1) `SApp` $(s2) |]
  P.TTuple ts ->
    case ts of
      (x1 : x2 : xs) -> do
        let s1 = typeToSingExp x1
            s2 = typeToSingExp x2
            base = [| $(s1) `SPair` $(s2) |]
        foldr (\x acc -> [| STup $(typeToSingExp x) $(acc) |]) base xs
      _ -> fail $ "Cannot convert tuples of " ++ show (length ts) ++ " elements"
```

Listing 31: Converting CloFRP types to Haskell singleton expressions.

Meta-programming in Haskell is in fact quite pleasant thanks to the *expression quotations* which are written inside oxford brackets `[|...|]`. We need to manually create the quotation for type-level string literals, but otherwise it is pretty easy to just wrap the concrete syntax in the brackets and splice with `$(...)`.

Finally, we can associate a CloFRP program with its type at compile-time using the `CloFRP` type shown in Listing 32.

```haskell
-- |A CloFRP program of a type, executed in an environment
data CloFRP :: CloTy -> * -> * where
  CloFRP :: EvalRead a -> P.Expr a -> Sing t -> CloFRP t a
```

Listing 32: Haskell representation of CloFRP programs at compile-time.

The `CloFRP` type is indexed over types of the `CloTy` kind. It consists of an environment in which evaluate an expression, along with a singleton that determines its `CloTy` index. Concretely, constructing values of `CloFRP` happens at compile time using the `clofrp` quasi-quoter, shown in Listing 33.

```
clofrp :: QuasiQuoter
clofrp = QuasiQuoter
  { quoteExp  = quoteclofrp
  , ...
  } where
    quoteclofrp :: String -> Q Exp
    quoteclofrp s = do
      prog <- liftParse P.prog s
      checkProgQExp prog
      (expr, ty, rd) <- progToEvalQExp prog
      let sing  = typeToSingExp ty
      let exprQ = liftData expr
      let rdQ   = liftData rd
      [|CloFRP $(rdQ) $(exprQ) $(sing)|]
```

Listing 33: The `clofrp` quasi-quoter.

The quasi-quoter can be used by a Haskell programmer simply by wrapping an expression in `[clofrp|...|]`, for example:

```
clofrp_const = [clofrp|
  const : forall (k : Clock) a. a -> Stream k a.
  const = \x -> fix (\g -> fold (Cons x g)).
  main : Stream K0 Unit.
  main = const MkUnit.
|]
```

At compile time, the syntax inside the brackets is passed to `quoteclofrp` as a string. We then parse it to an AST and type-check it, before preparing it for evaluation, which yields a main expression `expr`, a CloFRP type `ty` and an environment `rd` to execute the expression in. `typeToSingExp` to gives us a Haskell expression that encodes the singleton type of the CloFRP program above. We can then use `liftData` to lift Haskell values to their expressions. Using Haskell's support for generic programming, we get `liftData` completely for free, since we can derive it when we declare the data-types for our AST[4]! Finally, the `CloFRP` expression is constructed. After template-expansion, Haskell will in fact be able to infer the type of `clofrp_const` which is `'CTFree "Stream" ':@: 'CTFree "K0" ':@: 'CTFree "Unit"`. Here `K0` is the CloFRP equivalent of the clock-constant κ_0 from [4].

Now that we can know the type of a CloFRP program at compile time, we can ask ourselves how to marshall values between CloFRP and Haskell. Since both CloFRP and Haskell have an open type-system, we cannot create a closed algorithm to marshall the types. Therefore, we use type-classes to let the user describe how to convert the types that he cares about.

Concretely, we declare two type-classes, seen in Listing 34.

```
class ToHask (t :: CloTy) (r :: *) | t -> r where
  toHask :: Sing t -> Value a -> r

class ToCloFRP (r :: *) (t :: CloTy) | t -> r where
  toCloFRP :: Sing t -> r -> Value a
```

Listing 34: Type-classes for marshallling CloFRP values.

`ToHask` describes how to convert a CloFRP type `t` to an inhabited Haskell type `r`. The `t -> r` syntax is a `FunctionalDependency` which makes Haskell check that `t` alone will always determine which `r` to use when searching for type-class instances of `ToHask`.

[4]We have left out the deriving clauses throughout our code snippets for brevity.

Similarly, `ToCloFRP` descibes the inverse transformation. While it will be up to the user to provide instances that covers his use cases, we can give a few general instances, such as pairs:

```
instance (ToCloFRP h1 c1, ToCloFRP h2 c2) => ToCloFRP (h1, h2) ('CTTuple [c1, c2]) where
  toCloFRP (SPair s1 s2) (x1, x2) = Tuple [toCloFRP s1 x1, toCloFRP s2 x2]
  toCloFRP s             (x1, x2) = error ...

instance (ToHask c1 h1, ToHask c2 h2) => ToHask ('CTTuple [c1, c2]) (h1, h2) where
  toHask (SPair s1 s2) (Tuple [x1, x2]) = (toHask s1 x1, toHask s2 x2)
  toHask (SPair s1 s2) v                = error ...
```

These definitions describe converting from and to CloFRP and Haskell pairs, given that we know how to convert both of its components. These definitions also show the limits of the type-safety that we can recover. There is no way to convince the compiler that the first clause is enough, because we cannot link CloFRP `Value`s to their types. There are ways to achieve more type safety though, but they would probably involve quite a bit more work, and it would be hard to completely ensure safety across the language boundaries, as is quite often the also case when looking at the foreign function interfaces of many modern real-world languages.

We can now define functions for working with `CloFRP` values, shown in Listing 35.

```
execute :: (Pretty a, ToHask t r) => CloFRP t a -> r
execute (CloFRP er st expr sing) = toHask sing $ runEvalMState (evalExprCorec expr) er st

transform :: (Pretty a, ToCloFRP hask1 clott1, ToHask clott2 hask2)
          => CloFRP (clott1 :->: clott2) a -> hask1 -> hask2
transform (CloFRP er st expr (SArr s1 s2)) input = toHask s2 $ runEvalMState (evl expr) er st where
  evl e = do
    Closure cenv nm ne <- evalExprStep e
    let inv = toCloFRP s1 input
    let cenv' = extendEnv nm inv cenv
    withEnv (combine cenv') $ evalExprCorec ne
```

Listing 35: Simple functions for working with `CloFRP` values.

`execute` simply executes any CloFRP program that produces a CloFRP value of type `t` and returns a Haskell value of type `r`, given that such a marshalling exists. As such, CloFRP programs along with `execute` make for a good way to safely write productive Haskell programs. `transform` on the other hand, can also take a Haskell value as input. `transform`'s type signature is very uplifting, as it promises that we can essentially write safe, productive definitions in CloFRP and use them to consume and produce Haskell values almost seamlessly. Unfortunately, it is not so simple. Because the input value is stored in the evaluation environment, `transform` will inevitably lead to a space leak if the input Haskell value is infinite, as there will remain a pointer to the head of the input until `transform` terminates.

We can deal with this issue for specific types of input. We will focus on streams (Haskell's lists) here, since that is the canonical data structure for functional reactive programming. What we need to do is not simply put the input value in the evaluation environment. Instead, the input will in fact determine the evaluation – since we are working with causal streams in this instance, there will be one element in the output stream for every element in the input stream. As such, the input stream determines how the "clock" ticks. Listing 36 shows how we encode such a transformation.

```haskell
streamTrans :: (Pretty a, ToCloFRP hask1 clott1, ToHask clott2 hask2, KnownSymbol k)
            => CloFRP ('CTFree "Stream" ':@: 'CTFree k ':@: clott1
                       ':->:
                       'CTFree "Stream" ':@: 'CTFree k ':@: clott2) a
            -> [hask1] -> [hask2]
streamTrans (CloFRP er st expr ((_ `SApp` s2) `SArr` (_ `SApp` s5))) input = do
  fromCloFRPStream $ runEvalMState (begin input) er st
  where
    begin xs = do
      Closure env nm e@(A ann _) <- evalExprStep expr
      let e' = P.substExpr (P.Prim P.Input) nm e
      let inputs = map (makeInput ann) xs
      withEnv (const env) $ evalExprOver inputs e'

    makeInput ann z =
      Fold $ Constr "Cons" [ toCloFRP s2 z,
                             TickClosure mempty "_" $ A ann $ P.Prim P.Input
                           ]

    fromCloFRPStream (Fold (Constr "Cons" [v, c])) = toHask s5 v : fromCloFRPStream c
    fromCloFRPStream v = error $ "fromCloFRPStream:" ++ pps v

evalExprOver :: forall a. Pretty a  => [(Value a)] -> P.Expr a -> EvalM a (Value a)
evalExprOver f = foldr mapping (const $ pure $ runtimeErr "End of input") f where
  mapping x accfn expr = do
    v <- withInput x $ evalExprStep expr
    case v of
      Fold (Constr "Cons" [y, TickClosure cenv nm e]) -> do
        cont <- withEnv (const $ extendEnv nm (Prim Tick) cenv) $ accfn e
        pure $ Fold $ Constr "Cons" [y, cont]
      _ -> error (pps v)
```

Listing 36: The `streamTrans` combinator for functional reactive programming in CloTT.

The code is rather convoluted, unfortunately. The type of `streamTrans` says that:

1. Its first argument is a `CloFRP` program that, on any clock `k`, consumes a guarded stream of `hask1` values and produces streams of `hask2` values.
2. Its second argument is an input list of `hask1` values.
3. It returns a list of `hask2` values.

Most of the work is done by `evalExprOver` which evaluates a CloFRP expression one step every time it receives a value from the input stream. References to input values are treated specially in the expression to be evaluated. An expression created with the `Input` constructor always points to the "current" input value. The current input value is always exactly the ith value in the input stream after going under i tick-abstractions in the CloFRP programs.

There is certainly room for improvements in this area of the project – the interoperation could be more type-safe, and evaluating CloFRP programs in a Haskell context could benefit from a more principled approach. Haskell does not distinguish between guarded-recursive and (co)inductive types, and therefore it is not always obvious how the evaluation of a CloFRP program that depends on Haskell input should proceed. Ideally, we could design an API that would let the user elegantly assert these properties about their Haskell input, but it is not immediately apparent at this time how such an API would look.

8 Memory usage of CloFRP

This section will show some example programs to give the reader an impression of what kinds of programs we can write and execute in CloFRP. We shall examine how they execute by profiling their memory usage, in order to assess if the operational semantics and its implementation is suitable for a FRP language.

8.1 Positive integers

The Haskell program in Listing 37 uses the `clofrp` quasiquoter to output a million integers, starting from zero.

```
let nats = [clofrp|
  data StreamF (k : Clock) a f = Cons a (|>k f).
  type Stream  (k : Clock) a   = Fix (StreamF k a).
  nats : forall (k : Clock). Int -> Stream k Int.
  nats = fix (\g n -> fold (Cons n (\\(af : k) -> g [af] (n + 1)))).
  main : Stream K0 Int.
  main = nats 0.
|]
let n = 1000000
putStrLn . show $ take n $ execute nats
```

Listing 37: A million ascending integers.

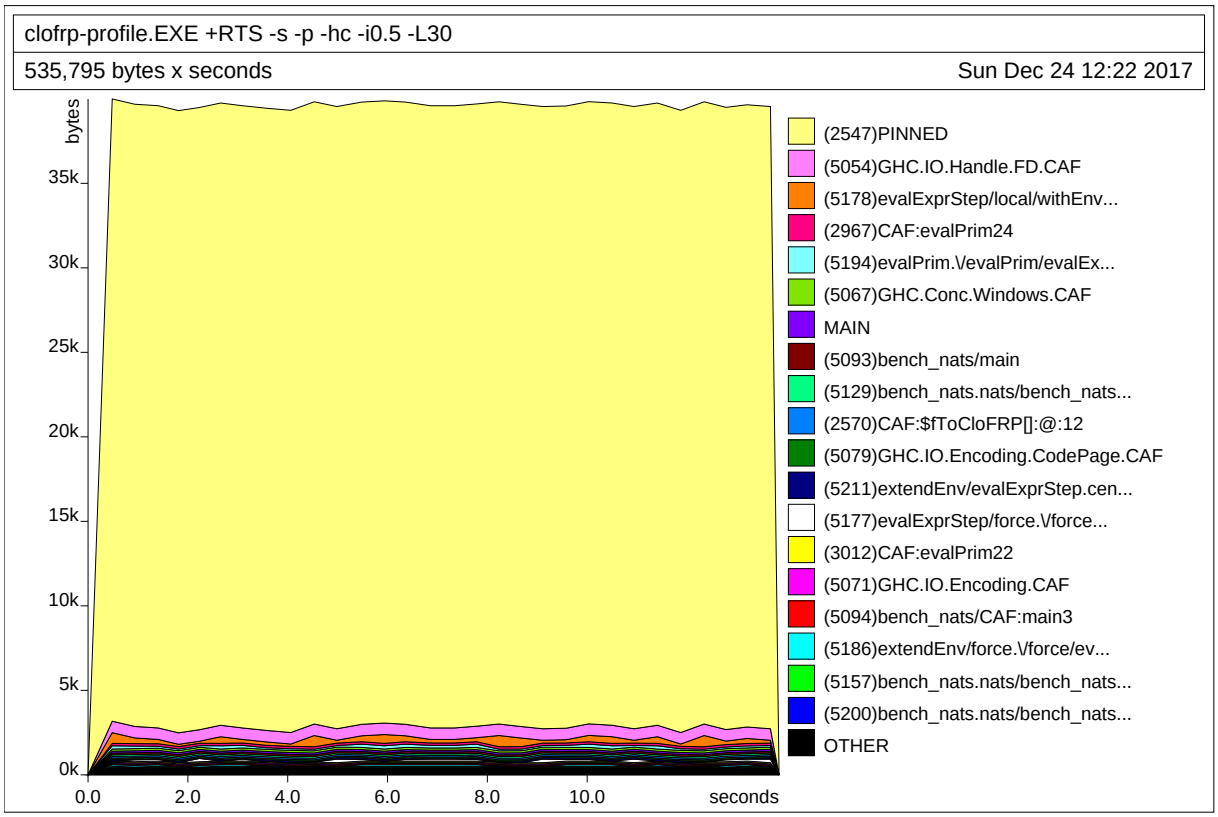


Figure 35: Memory profile of the program from Listing 37.

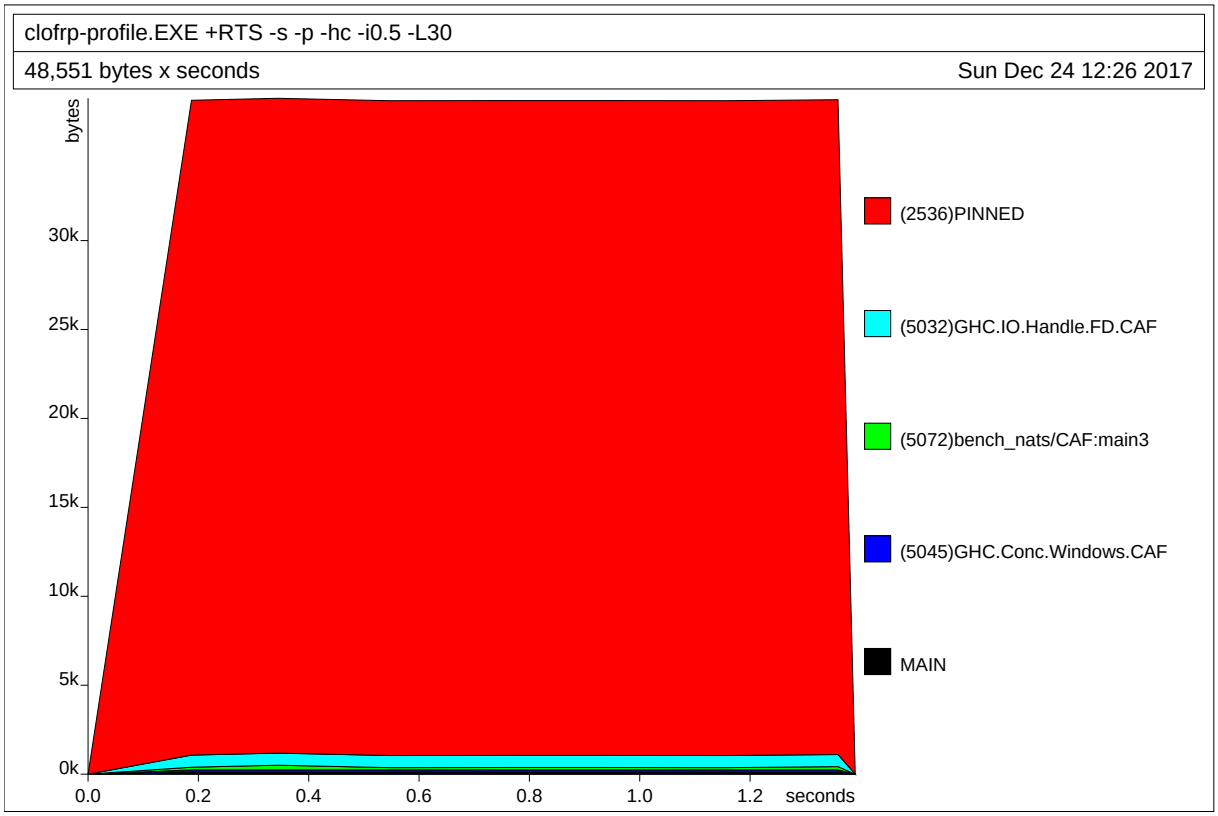


Figure 36: Memory profile of the Haskell program `putStrLn . show $ take n $ [0..]`.

As Figure 35 shows, the program executes in about 14 seconds in constant memory, which is good news! For comparison, Figure 36 shows the profile of the equivalent Haskell program (which is just one line). As expected, Haskell runs an order of magnitude faster. The good news is that our interpreter interprets the program in constant space, as expected.

In Haskell, a different way of generating positive integers is with the corecursive definition `nats = 0 : map (+1) nats`. In [4], Bahr, Grathwohl and Møgelberg encodes a similar definition in CloTT and we can do the same in CloFRP as shown in Listing 38.

```
let nats = [clofrp|
  data StreamF (k : Clock) a f = Cons a (|>k f).
  type Stream (k : Clock) a = Fix (StreamF k a).
  map : forall (k : Clock) a b. (a -> b) -> Stream k a -> Stream k b.
  map = \f -> fix (\g xs ->
    let Cons x xs' = unfold xs
    in  fold (Cons (f x) (\\(af : k) -> g [af] (xs' [af])))
  ).
  nats : forall (k : Clock). Stream k Int.
  nats = fix (\g -> fold (Cons 0 (\\(af : k) -> map (\x -> x + 1) (g [af])))).
  main : Stream K0 Int.
  main = nats.
|]
let n = 2000
putStrLn . show $ take n $ execute nats
```

Listing 38: Positive integers translated from [4].

While an elegant encoding theoretically, this definitions poses more trouble for the CloFRP evaluator,

which leaks memory as shown in Figure 37.

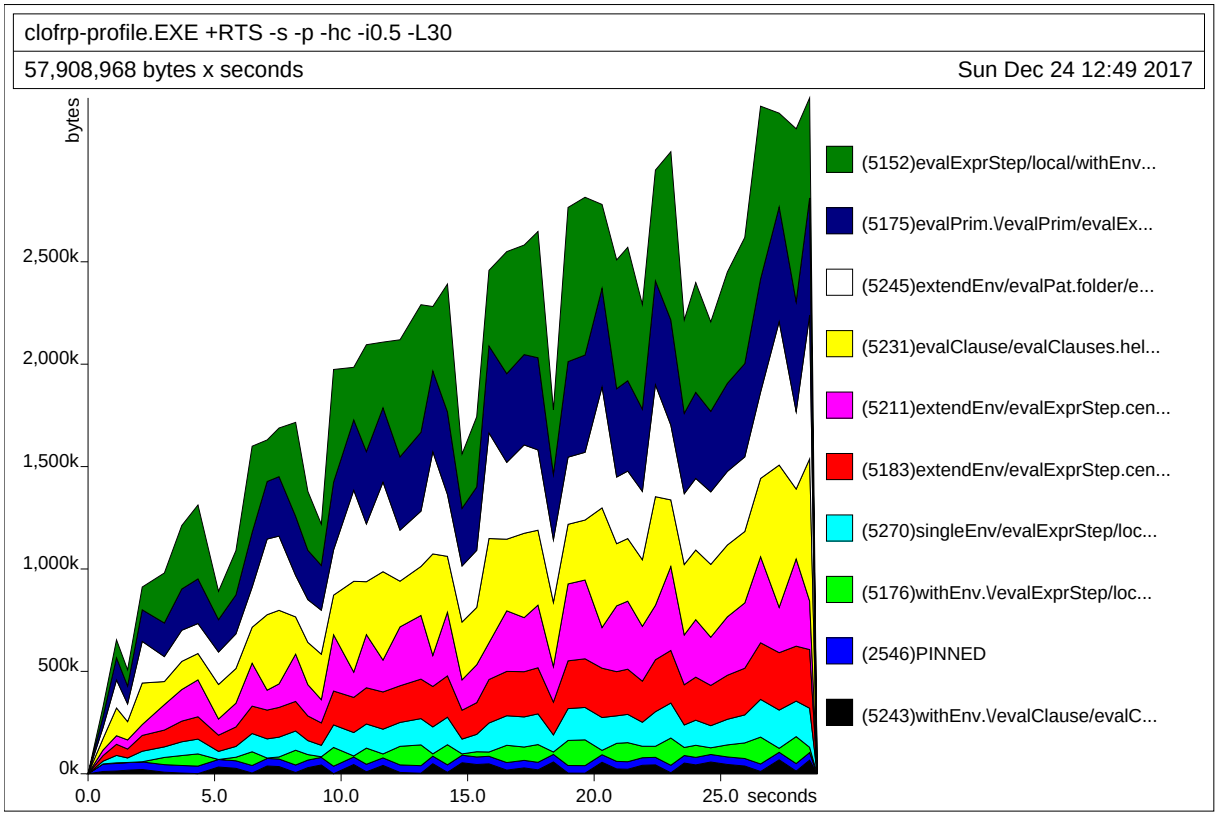


Figure 37: Memory profile of the program in Listing 38.

Haskell's fixpoint operator is defined as `fix f = let x = f x in x`, which works because Haskell is lazily evaluated. `x` will just be a pointer to the expression `f x` which will chase itself circularly. As Haskell uses call-by-need semantics, the pointer always points to the most "up-to-date" result of the recursive computation. If you re-formumulate Haskell's `fix` as `fix f = f (fix f)`, you will actually see the same memory leak if you implement `nats` with `fix`, since Haskell does not memoize function applications. We shall talk more about call-by-need versus call-by-name semantics in Section 9.

8.2 Circular traversals of trees

Profiling the circular `replaceMin` program in Listing 6 gives the result seen in Figure 38.

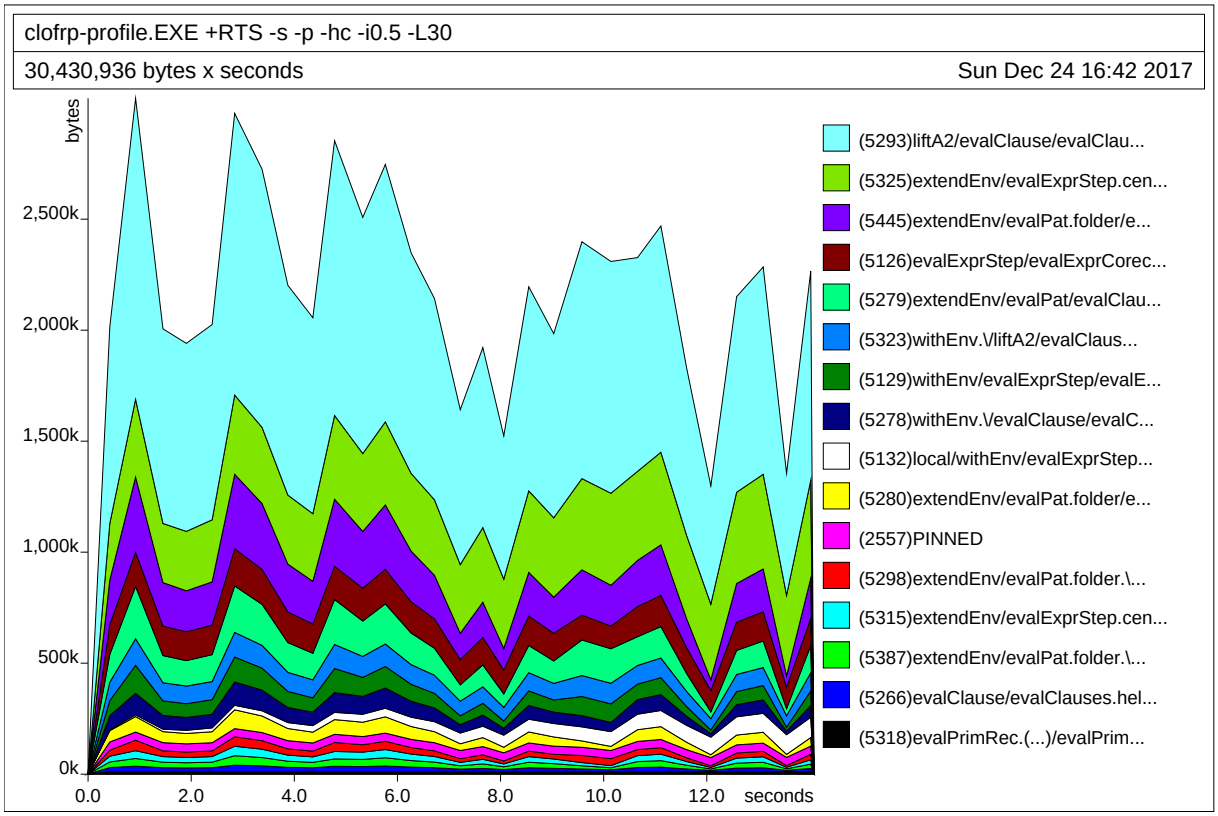


Figure 38: Memory profile of the `replaceMin` program from Listing 6.

While the program does use quite a bit of memory, it does not appear to be leaking any, which is good. This example shows that, in addition to infinite productive programs, clock-quantification can also be used to safely write circular lazy programs that still terminate and evaluate usefully.

8.3 Evaluating coinductive streams

Our evaluation algorithm can also evaluate coinductive streams without problems. The every-other function takes a stream of values and returns every other. Such a function is trivial to implement in Haskell, but is frustratingly not implementable with standard guarded recursion, as it is not causal. However, as shown in [4], it *is* implementable when adding clock-quantification to the mix. We can also implement it in CloFRP, as seen in Listing 39.

```
cos : forall (k : Clock) a. a -> |>k (CoStream a) -> CoStream a.
cos = \x xs ->
  Cos (fold (Cons x (\\(af : k) -> uncos (xs [af])))).

eofix : forall (k : Clock) a. |>k (CoStream a -> CoStream a) -> CoStream a -> CoStream a.
eofix = \f xs ->
  let tl2 = tl (tl xs) in
  let dtl = (\\(af : k) -> (f [af]) tl2) in
  cos (hd xs) dtl.

eo : forall a. CoStream a -> CoStream a.
eo = fix eofix.

truefalse : forall (k : Clock). Stream k Bool.
truefalse = fix (\g -> cons True (\\(af : k) -> cons False g)).

main : Stream K0 Bool.
main = let Cos xs = eo (Cos truefalse) in xs.
```

Listing 39: The every-other function implemented in CloFRP.

Listing 39 implements the `eo` function, which takes every other element of a coinductive stream, and uses it to generate an infinite stream of `Trues` from an input stream that oscillates between `True` and `False`. While it is a contrived example, it shows that it is possible to type-check and evaluate coinductive streams in CloFRP. Figure 39 shows the memory profile of the program when terminated after evaluating one million output values.

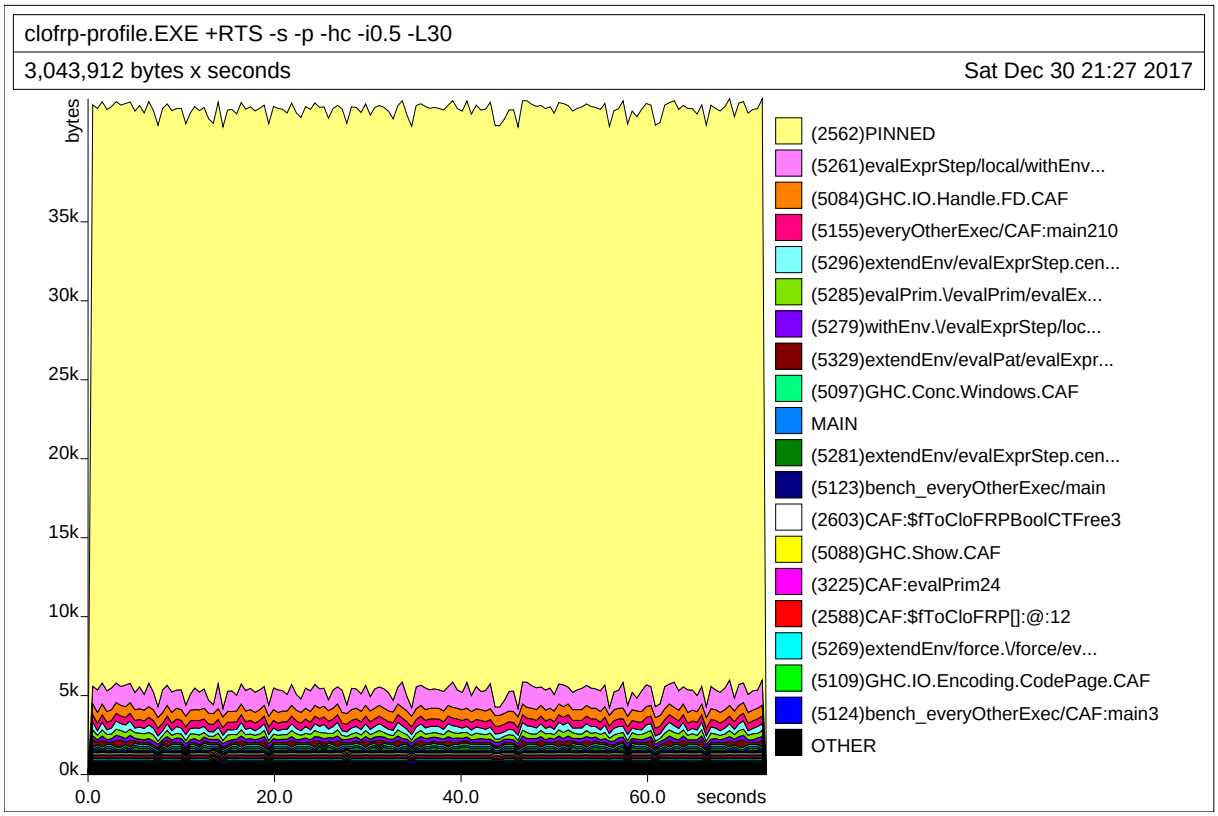


Figure 39: Memory profile of Listing 39

8.4 Evaluating stream transformers

The examples we have seen so far produce output that is converted to Haskell. We have not yet seen how CloFRP evaluates FRP programs that take input from Haskell. Listing 40 shows a simple program that simply takes a stream of pairs of random integers and adds them together. Figure 40 shows its memory profile which reveals that it does not leak memory.

```
clofrp_add = [clofrp|
  -- |>k is applicative
  app : forall (k : Clock) a b. |>k (a -> b) -> |>k a -> |>k b.

  main : Stream K0 (Int, Int) -> Stream K0 Int.
  main =
    fix (\g pairs ->
      case unfold pairs of
        | Cons pair xs ->
          case pair of
          | (x1, x2) -> fold (Cons (x1 + x2) (app g xs))
    ).
|]
main :: IO ()
main = do
  let n = 800000
  g <- newStdGen
  let inputs = randoms g `zip` randoms g
  let output = take n (streamTrans clott_add_int inputs)
  putStrLn . show $ output
```

Listing 40: CloFRP program that simply adds two random integers together.

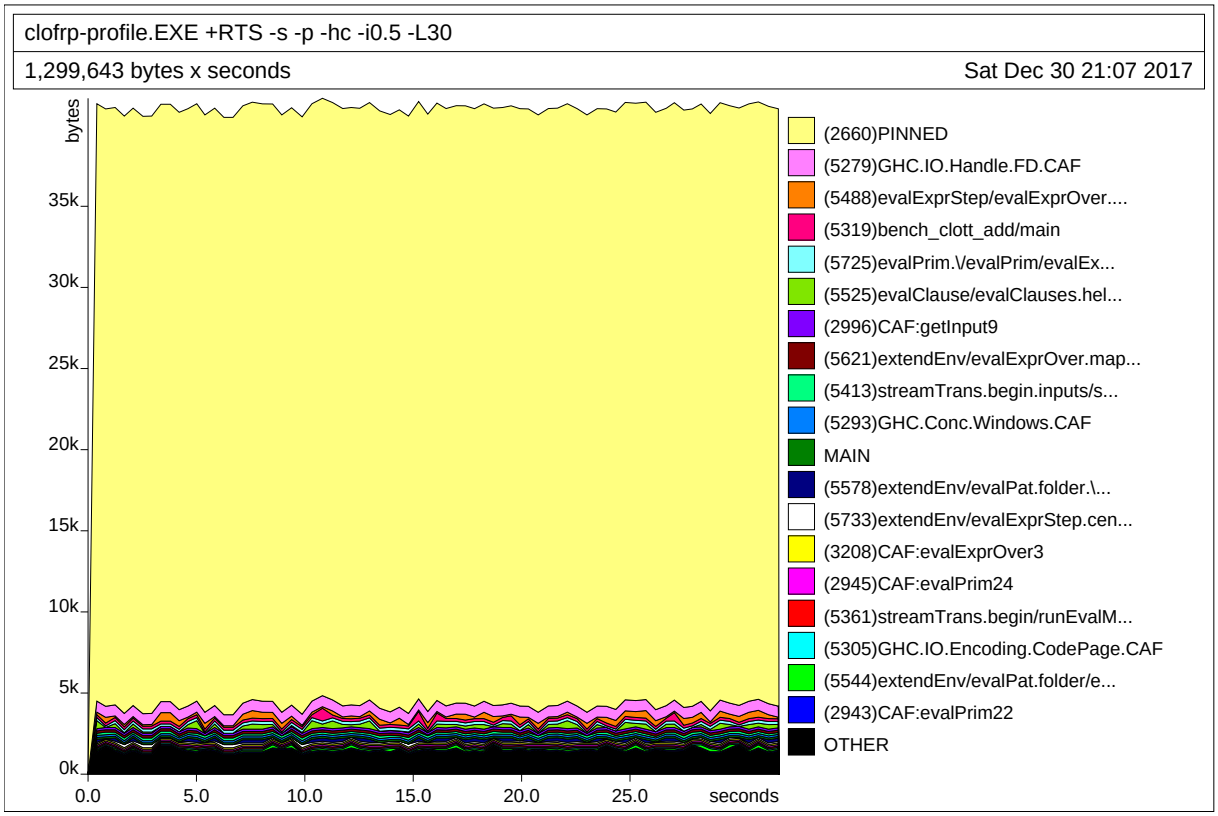


Figure 40: The memory profile of the program in Listing 40.

8.5 Guarded binary trees

So far, the streams are the only guarded recursive data-type we have seen. But we can also guard other data-types, for example binary trees.

```
1  data BinTreeF (k : Clock) a f = Branch a (|>k f) (|>k f).
2  type BinTree (k : Clock) a = Fix (BinTreeF k a).
3  data Unit = MkUnit.
4
5  main : BinTree K0 Unit.
6  main =
7    let constBinTree = \x ->
8          fix (\g -> fold (Branch x g g))
9    in constBinTree MkUnit.
```

Listing 41: Guarded binary trees in CloFRP.

Listing 41 shows how we can encode guarded binary trees in CloFRP. The main function, when evaluated, generates an infinite binary tree where every node just contains `MkUnit`. Figure 41 shows its memory usage when it is evaluated to a depth of 22.

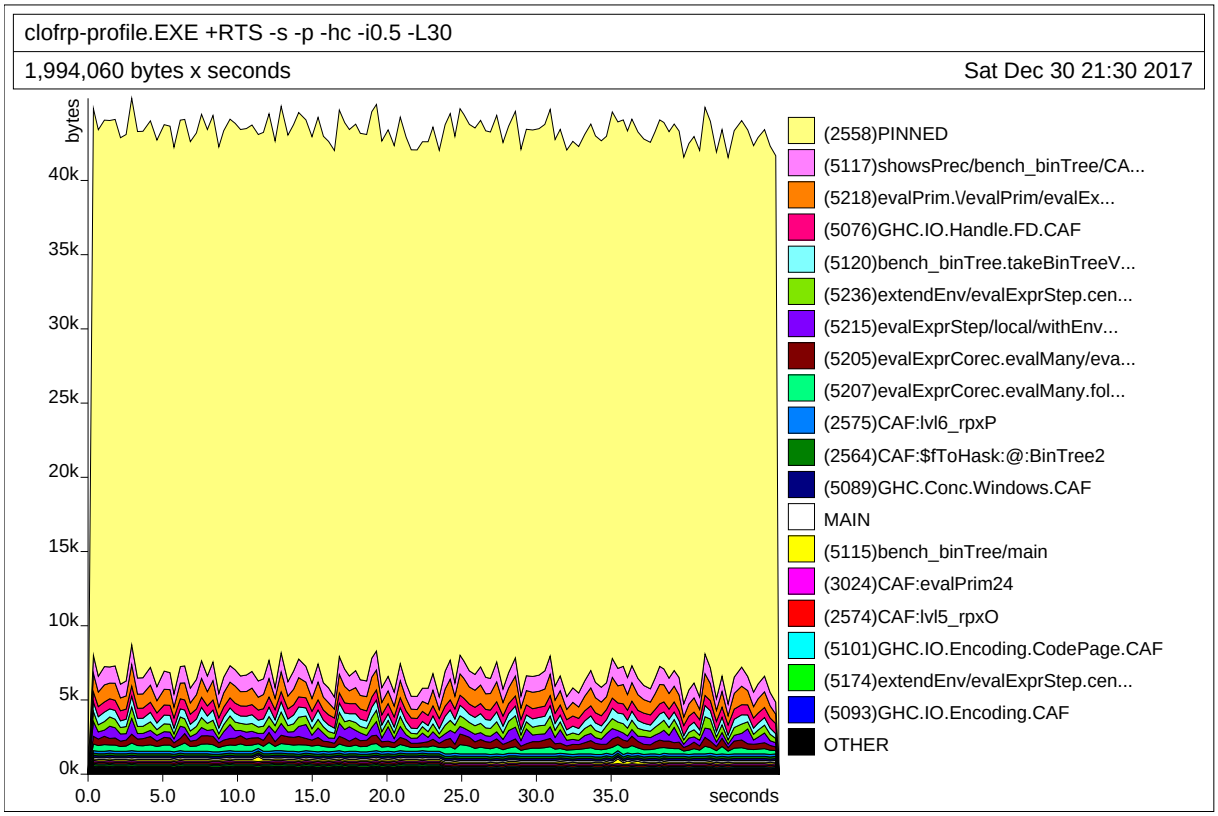


Figure 41: The memory profile of the binary tree program in Listing 41.

8.6 Higher-order streams

As noted in the beginning of this report, traditional FRP systems have struggled with representing higher-order streams in a memory-safe way. We can write programs that deal with higher-order streams in CloFRP easily, as demonstrated by the program shown in Listing 42.

```
1  const : forall (k : Clock) a. a -> Stream k a.
2  const = \x -> fix (\g -> cons x g).
3
4  nats : forall (k : Clock). Int -> Stream k Int.
5  nats = fix (\g n -> cons n (\\(af : k) -> g [af] (n + 1))).
6
7  main : Stream K0 (Stream K0 Int).
8  main = const (nats 0).
```

Listing 42: The constant stream of streams of natural numbers.

Figure 42 shows how the program in Listing 42 evaluates with respect to memory usage.

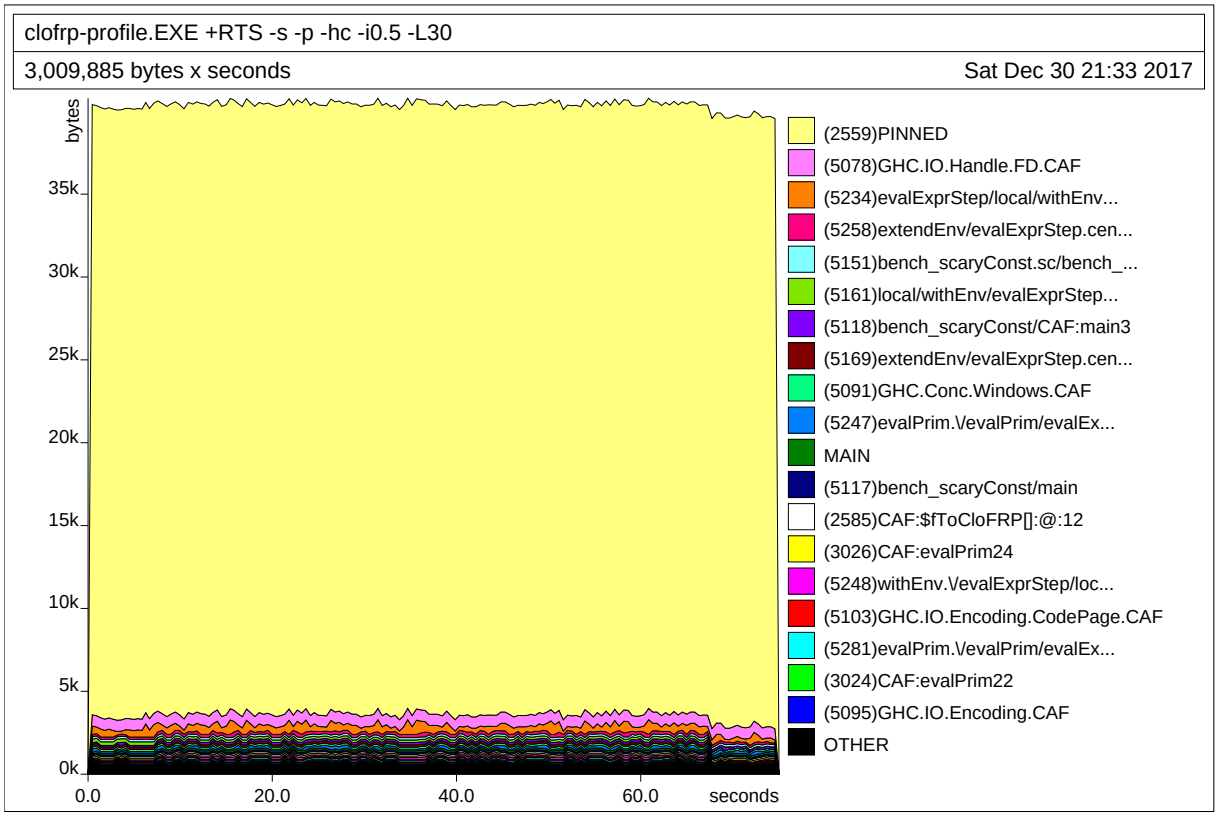


Figure 42: The memory profile of the higher-order stream program in Listing 42.

Interestingly, Listing 42 shows an encoding of the `bad_const` program from [2], which inherently leaks memory. However, the memory profile in Figure 42 shows constant memory usage! How can this be? The truth is that since CloFRP is evaluated through a *call-by-name* strategy and *not* through a *call-by-need* strategy, no memory will leak as we do not store the stream through time. Instead, it is re-evaluated constantly. This works fine in this contrived example, but is not feasible with expensive streams that are not generated deterministically, e.g. user input. We shall talk more about this in the discussion in Section 9.

9 Discussion and conclusion

This section will reflect on the project as a whole in relation to the problem statement: *The goal of this project is to produce a practical implementation of the simply-typed version of the* CloTT *[4] language, primarily targeted at the domain of functional reactive programming, in order to explore how the theory of* CloTT *integrates with a practical setting.*

To this end, this we have made the following contributions:

- We have designed and implemented a Haskell-embedded programming language based on the simply-typed fragment of CloTT. The language allows programmers to declare their own types via Algebraic Data-Type declarations and type-synonyms ála Haskell.
- We have designed and implemented a state-of-the-art type-inference system that can infer higher-rank polymorphic types, based on the work of Dunfield and Krishnaswami [30], but which also supports impredicative instantiation of type-variables through explicit type-applications.
- We have designed a big-step operational semantics for evaluating productive programs that oscillates between call-by-value and call-by-name evaluation.

- We have implemented the semantics as an interpreter in Haskell, and designed a system that facilitates somewhat type-safe interaction between Haskell and CloFRP.
- Finally, we have demonstrated the implementation by executing selected CloFRP programs and profiling their memory consumption over time.

Of course, our contribution is not perfect, and has at least the following issues:

- The lack of a proper type-class system (both for type-checking and evaluation) excludes many forms of higher-order abstraction.
- The type-checker does not do coverage checking, so we can easily introduce diverging programs by writing incomplete pattern matches.
- While some forms of impredicativity is expressible, it is not enough to be a practically useful feature of the language.
- The operational semantics uses a call-by-name strategy to evaluate guarded fixpoints. While this effectively rules out many space leaks, it sacrifices performance in such a way that it prohibits some elegant corecursive definitions. Additionally, non-deterministic values such as streams of user-input cannot be modeled with this strategy alone.
- The API for Haskell interoperation is somewhat heavy-handed and does not provide very strong compile-time guarantees.

As this project is of an exploratory nature, there are no grand conclusions to present. Instead, the work has revealed some interesting observations on the implementation of programming languages for safe functional reactive and corecursive programming:

- **Clock variables** can be modeled as a special case of normal type variables. Consequently, they can be inferred in most cases, and would be easy to integrate into existing languages as a standalone concept. Of course, the rest of the associated machinery, like guarded recursion, is a somewhat different beast to implement in existing languages, largely because it limits expressivity compared to general recursion (but provides more safety).
- **Simply-typed** CloTT is (unsurprisingly) a lot simpler to implement and specify compared to its dependently typed formulation. Specifically, since we do not normalize terms under binders, the typing rules related to "stopping" recursive evaluation are significantly simplified. The typing rule for tick-application in CloTT is rather complicated, and relies on guessing a clock variable κ'. In CloFRP, it is much simpler. The operational semantics rule for tick-application TickApp is also very simple, and always reduces, since we can be sure that we are always evaluating on the "first level" of a program.
- **Bidirectional type-inference** systems based on [30] work well with explicit type-applications. This is interesting in conjunction with impredicative polymorphism, since Haskell has currently abandoned its support for impredicative polymorphism, yet plans to re-implement it using explicit type-applications [39]. This was not an original goal of this project, and so has not been explored very much, but it appears that it would not be too difficult to reach some level of usefulness without incurring significant extra complexity.
- **Clock-quantification** effectively extends the expressiveness of guarded recursion, allowing it to model corecursive definitions that are not causal, also in a practical setting. While causality is a desirable property when dealing with time-based data, it is too restrictive to be useful for general programming. The mix of guarded recursion, clock-quantification and primitive recursion, gives a far more expressive language. Other forms of well-founded recursion could easily be added, or a termination checker could be implemented, to expand the range of programs that can be expressed in CloFRP. Specifically, clock-quantification integrates well with existing frameworks for type-safe programming, and therefore practical programs can be formulated with clock-quantification while still retaining significant programmer ergonomics.

9.1 Comparison with simple-frp

Now that we have discussed some of the aspects of this project, we can briefly compare it with our previous implementation [1] of the FRP calculus that Krishnaswami developed in [2], which we called ModalFRP, and described in Section 2.3. Notably, ModalFRP prevents the programmer from introducing space-leaks caused by remembering infinite values (unless you really, really, want to), by using a particular two-step operational semantics. ModalFRP does not feature clocks or ticks on them, but instead delayed values that are suspended on a heap. These delayed values are similar to tick-abstractions in many ways, except that only one (implicit) clock exists.

ModalFRP switches between evaluating fixpoints one "step" using a call-by-value semantics, and a "tick" semantics that forces all suspended computations on the heap, and deletes all previous allocations. It has a typing discipline that makes sure that deleted pointers are not referenced.

Our semantics is similar to ModalFRP in many ways, but does *not* feature a heap. When ModalFRP's interpreter encounters a delayed expression, it allocates its body on the heap and returns a pointer to it. In CloFRP, we instead directly return a tick-abstraction. Therefore, our tick-semantics models a *call-by-name* strategy instead of ModalFRP's *call-by-need* strategy. Call-by-name is significantly simpler to implement for CloFRP, as it is not possible to determine what thunks to de-allocate without a garbage collector, in contrast to ModalFRP. On the other hand CloFRP is more expressive and applicable outside recursion over time-dependent values, since it also features productive corecursion through clock-quantification. Call-by-name semantics is sometimes preferable to call-by-need, since it causes fewer space leaks, as seen in Section 8.6. On the other hand, non-computational temporal streams, such as streams representing user-input, are not expressible, and some corecursive definitions evaluate prohibitively slowly, such as `nats` in Listing 38.

Cave et al. [14] expanded on the work of Krishnaswami by giving a similar calculus and semantics that also enforces liveness properties, particularly fairness of schedulers. Their calculus separates inductive types $\mu X.\, F$ and co-inductive types $\upsilon X.\, F$ and they provide iteration and co-iteration operators to traverse them. This separation along with the absense of a fixpoint combinator allows them to express fine-grained distinctions on the behaviour of programs in their types. For example, they can express the "eventually" modality of linear temporal logic that *guarantees* that a certain event will happen at some point. CloTT is not capable of this, since the least- and greatest-fixpoints of type-constructors are collapsed, and so we cannot distinguish between "will-happen" and "might-happen" temporally recursive types. On the other hand, the calculus in [14] suffers from many of the same programmer-ergonomic nuisances that Krishnaswami's calculus from [2] does, and also cannot express non-causal productive programs, which limits its applicability to time-dependent domains.

In conclusion, this projects shows that simply-typed CloTT can be implemented using existing tools and techniques for building modern practical programming languages. While this implementation is still academia-quality software, and is still missing some crucial features, it does indicate that clock-quantification and guarded recursion is a viable route for type-safe co-programming in a practical setting.

10 Future Perspectives

There are still several features that CloFRP needs before being an actually practical language to program in; as previously mentioned, a coverage-checker and a type-class system are the most important omissions.

The biggest design issues with the current implementation are with the interpreter and the Haskell-integration. There are ways to improve the situation, but a more promising angle would perhaps be to pursue a denotational semantics which interprets CloFRP programs as Haskell programs instead. This would solve both problems in one strike, and we avoid constructing a complicated runtime with e.g. garbage collection. Experience from previous projects [1] indicate that this is a promising direction to pursue.

Much of the complexity of CloTT disappeared when considering its simply-typed fragment, but so did much of its utility. It is particularly in a dependently typed setting that dealing with coinductive data is currently difficult, so extending this implementation to the full CloTT specification would be interesting.

Finally, an unanticipated discovery of this project was that explicit type-applications in combination with our inference system enables impredicative polymorphism through a simple mechanism. This could be an interesting avenue of research to pursue that is entirely orthogonal to guarded recursion and clock-quantification.

References

[1] A. Schønemann, "Resource-safe functional reactive programming." unpublished semester project, available online at https://drive.google.com/open?id=1vW6Xy2j-AAavh0Xx4gMqMxmI7CI18PVi, 2017.

[2] N. R. Krishnaswami, "Higher-order reactive programming without spacetime leaks," in *International Conference on Functional Programming (ICFP)*, Sept. 2013.

[3] R. Atkey and C. McBride, "Productive coprogramming with guarded recursion," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, (New York, NY, USA), pp. 197–208, ACM, 2013.

[4] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg, "The clocks are ticking: No more delays! reduction semantics for type theory with guarded recursion," in *Logic in Computer Science (LICS), 2017 32snd Annual ACM/IEEE Symposium on*, ACM/IEEE, 2017.

[5] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, (New York, NY, USA), pp. 263–273, ACM, 1997.

[6] E. Czaplicki, "Elm: Concurrent frp for functional guis," *Senior thesis, Harvard University*, 2012.

[7] Y. H. Group, "Yampa," 2003.

[8] T. C. development team, "The coq proof assistant," 2004.

[9] Z. Wan, W. Taha, and P. Hudak, "Event-driven frp," in *International Symposium on Practical Aspects of Declarative Languages*, pp. 155–172, Springer, 2002.

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

[11] H. Nakano, "A modality for recursion," in *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*, pp. 255–266, IEEE, 2000.

[12] U. Norell, *Towards a practical programming language based on dependent type theory*, vol. 32. Citeseer, 2007.

[13] N. R. Krishnaswami and N. Benton, "Ultrametric semantics of reactive programs," in *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pp. 257–266, IEEE, 2011.

[14] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka, "Fair reactive programming," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 361–372, ACM, 2014.

[15] 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*, POPL '12, (New York, NY, USA), pp. 45–58, ACM, 2012.

[16] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal, "Guarded dependent type theory with coinductive types," *CoRR*, vol. abs/1601.01586, 2016.

[17] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal, "Programming and reasoning with guarded recursion for coinductive types," in *International Conference on Foundations of Software Science and Computation Structures*, pp. 407–421, Springer, 2015.

[18] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[19] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the american mathematical society*, vol. 146, pp. 29–60, 1969.

[20] D. Leijen, "Hmf: Simple type inference for first-class polymorphism," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, (New York, NY, USA), pp. 283–294, ACM, 2008.

[21] D. Vytiniotis, S. Weirich, and S. Peyton Jones, "Fph: First-class polymorphism for haskell," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, (New York, NY, USA), pp. 295–306, ACM, 2008.

[22] D. Vytiniotis, S. Weirich, and S. Peyton Jones, "Boxy types: Inference for higher-rank types and impredicativity," in *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, (New York, NY, USA), pp. 251–262, ACM, 2006.

[23] S. P. Jones, G. Washburn, and S. Weirich, "Wobbly types: type inference for generalised algebraic data types," tech. rep., Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.

[24] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *Journal of functional programming*, vol. 17, no. 1, pp. 1–82, 2007.

[25] D. Vytiniotis, S. P. Jones, T. Schrijvers, and M. Sulzmann, "Outsidein (x) modular type inference with local assumptions," *Journal of functional programming*, vol. 21, no. 4-5, pp. 333–412, 2011.

[26] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 1, pp. 1–44, 2000.

[27] T. Coquand, "An algorithm for type-checking dependent types," *Science of Computer Programming*, vol. 26, no. 1-3, pp. 167–177, 1996.

[28] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi, "A bi-directional refinement algorithm for the calculus of (co) inductive constructions," *arXiv preprint arXiv:1202.4905*, 2012.

[29] A. Abel, T. Coquand, and P. Dybjer, "Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory," in *Mathematics of Program Construction*, pp. 29–56, Springer, 2008.

[30] J. Dunfield and N. R. Krishnaswami, "Complete and easy bidirectional typechecking for higher-rank polymorphism," in *Int'l Conf. Functional Programming*, Sept. 2013. arXiv:1306.6032[cs.PL].

[31] A. Abel, "Termination checking with types," *RAIRO-Theoretical Informatics and Applications*, vol. 38, no. 4, pp. 277–319, 2004.

[32] G. Bierman, E. Meijer, and M. Torgersen, "Lost in translation: Formalizing proposed extensions to c#," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, vol. 42, pp. 479–498, 10 2007.

[33] B. Heeren, J. Hage, and S. D. Swierstra, "Generalizing hindley-milner type inference algorithms," tech. rep., Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, 2002.

[34] H. Barendregt, W. Dekkers, and R. Statman, *Lambda calculus with types*. Cambridge University Press, 2013.

[35] J. B. Wells, "Typability and type checking in system f are equivalent and undecidable," *Annals of Pure and Applied Logic*, vol. 98, no. 1-3, pp. 111–156, 1999.

[36] R. S. Bird, "Using circular programs to eliminate multiple traversals of data," *Acta informatica*, vol. 21, no. 3, pp. 239–250, 1984.

[37] D. Pattinson, P. Hancock, and N. Ghani, "Representations of stream processors using nested fixed points," *Logical Methods in Computer Science*, vol. 5, 2009.

[38] N. A. Danielsson and T. Altenkirch, "Subtyping, declaratively: An exercise in mixed induction and coinduction," in *Proceedings of the 10th International Conference on Mathematics of Program Construction*, MPC'10, (Berlin, Heidelberg), pp. 100–118, Springer-Verlag, 2010.

[39] S. Peyton Jones, "Ghc wiki: Impredicative polymorphism," 2017.