Productive Co-Programming in Haskell with Ticking Clocks

Adam Schønemann
Msc Graduate
IT University of Copenhagen
Denmark
adsc@itu.dk, adamschoenemann@gmail.com

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 (Clocked Type Theory). 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. This paper 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 a compilation scheme targeting Haskell. We conclude that productive practical coprogramming with clocks is realizable without sacrificing significant programmer ergonomics or introducing prohibitive complexity into the theory or the implementation of existing languages.

Keywords functional programming, guarded recursion, clock variables

1 Introduction

Lazy evaluation, as popularized by Haskell, allows us to express and compose infinite programs that still yield useful output. Under lazy evaluation, an infinite program is *productive* if it produces output for any finite "point in time". `zeros = 0 : zeros` is productive, while `xs = xs` is not. Non-productive programs are unresponsive, and correspond to a proof of *False* in the Curry-Howard interpretation, and therefore we wish to exclude such programs statically. Infinite programs can also be *causal*. A causal definition produces output at time n by examining input only from time m where $m \leq n$. For example, `noncausal (x : y : xs) = y : noncausal xs` will always depend on a future value before producing any output, and thus cannot produce output in the current time-step. If the input to `noncausal` is being constructed "at the same time" as `noncausal` produces output, then the program will hang. As a concrete example [1] from Atkey & McBride [1], consider the two following definitions of `map` on infinite streams in Haskell:

```
data Stream a = Cons a (Stream a)
map, maap :: (a -> b) -> Stream a -> Stream b
map f (Cons x xs) = Cons (f x) (map f xs)
maap f (Cons x1 (Cons x2 xs)) =
  Cons (f x1) (Cons (f x2) (maap f xs))
```

If applied to fully constructed streams, their behaviour is identical. But if applied to streams that are still "being constructed", `maap` can diverge while `map` executes as expected for any stream. For example:

```
data Stream a = Cons a (Stream a)
nats = Cons 0 (map (+ 1) nats)
badnats = Cons 0 (maap (+ 1) badnats)
```

`nats` produces the infinite stream of natural numbers, but `badnats` diverges, since `maap` requires two values to be available on the input stream but there ever only is one. Since `nats` and `badnats` has the same type, their distinct semantics is not captured adequately by their types.

1.1 Guarded Recursion

Guarded recursion was originally suggested by Nakano [9] to address the challenge of working with coinductive data in proof assistants. Guarded recursion introduces a new modality ▷, pronounced "later", which makes it possible to distinguish between values we can use now, and values we can only use later. 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". For example, the type of infinite streams would be expressed as `data Stream a = Cons a ▷(Stream a)` while the unrestricted fixpoint combinator is replaced with the guarded fixpoint `fix :: (▷a -> a) -> a`.

Several approaches for using guarded recursion for functional reactive programming have been proposed [4, 7, 8]. Since FRP deals primarily with time-dependent data, guarded recursion seems to be a natural fit for FRP.

Guarded recursion is a simple and effective solution for writing infinite yet productive (and causal) programs. The problem is that the world of guarded recursion is too restrictive for many domains. We cannot make finite observations on guarded recursive data; as such we cannot type the following definition:

PL'18, September 27–28, 2018, St. Louis, MO, USA
2018.

[1] Which McBride learned of from Nils Anders Danielsson in https://mazzo.li/epilogue/index.html%3Fp=186.html

```
take :: Natural -> Stream a -> List a
take 0 _ = []
take (n + 1) (Cons x xs) = x : take n xs
```

The problem arises because `xs` has type $\triangleright$(`Stream a`) in the second clause, whereas `take` expects its second argument to have type `Stream a`. The global time-slicing of guarded recursion prevents us from ever looking into the "future", even on fully-constructed data. This is by design, but once data has been constructed, we should be able to scrutinize its structure as far as desired. Atkey & McBride [1] solve this problem by *localizing* the conceptual time-line that data is being constructed on using the concept of *clock variables*.

1.2 Clock variables

A clock variable represents a time-line that can be used for constructing guarded recursive data. As such, the guardedness type constructor is now indexed by a clock variable κ. Consequently, we can conceptually re-define the type of guarded streams as

```
data Stream (κ :: Clock) (a :: *) = Cons a ▷^κ(Stream κ a)
```

The guarded fixpoint combinator is also indexed over a clock variable, and typed as

```
fix :: ∀(κ :: Clock). (▷^κa -> a) -> a
```

We can think of the guarded stream Stream κ a as being the type of "streams in construction", whereas the clock quantified type $\forall\kappa$. Stream κ a represents the type of "fully-constructed" streams. Intuitively, a clock represents some time-line that can be thought of as a resource to use for making finite observations on data. Thus, the type Stream κ a represents streams on which we can only observe the head "now", whereas the quantified type $\forall\kappa$. Stream κ a says that for *any* clock, we can get a stream, and thus we can safely destructure the stream as much as we would like. Atkey & McBride [1] also provide clock abstraction $\Lambda\kappa.e$ and clock application $e[\kappa]$ in the style of System F, as well as the type equality $\forall\kappa.\, A \equiv A$ when $\kappa \notin fv(A)$, and the primitive force $:: (\forall\kappa.\ \triangleright^\kappa A) \to (\forall\kappa.\, A)$. Using this, we can write

```
uncons :: (∀κ. Stream κ a) -> (a, ∀κ. Stream κ a).
uncons xs =
  ( Λκ. case xs[κ] of Cons x _ -> x
  , force (Λκ. case xs[κ] of Cons _ xs -> xs[κ]))
```

We can now type `take` for fully-constructed streams:

```
take :: Natural -> (∀κ. Stream κ a) -> List a
take 0 _ = []
take (n + 1) xs = x : take n xs' where
  (x, xs') = uncons xs
```

Using clock variables, we can only type `map` and `maap` above as

```
map :: ∀κ. (a -> b) -> Stream κ a -> Stream κ b
maap :: (a -> b) -> (∀κ. Stream κ a) -> (∀κ. Stream κ b)
```

Their types clearly reflect their behaviour, namely that `maap` is only defined for fully-constructed streams, which allows us to statically rule out programs such as `badnats`.

Bahr et al. [2] extended the work of AtKey & McBride to a dependent type-theory called CloTT and gave a strongly normalizing reduction semantics by introducing *ticks* on clocks $\alpha : \kappa$ and the *tick constant* $\diamond$, along with tick abstraction $\lambda(\alpha : \kappa).\ e$ and application $e[\alpha]$. The guarded type modality is extended to a dependent version as $\triangleright\,(\alpha : \kappa).\, A$. If α is not free in A we simply write $\triangleright^\kappa A$. A term of such a type can be forced by applying it to the tick constant $\diamond$, which in general is unsafe: $\lambda(x : \triangleright^\kappa A).\, x[\diamond]$ is for example not productive and should not be typeable. But it is safe to force an open term if κ is not free in the context, leading to the rule[2]:

$$\frac{\Gamma \vdash_{\Delta,\kappa} e : \triangleright\,(\alpha : \kappa).\, A \qquad \Gamma \vdash_{\Delta}}{\Gamma \vdash_{\Delta,\kappa} e[\diamond] : A[\diamond/\alpha]}$$

Here, the typing judgment $\Gamma_\Delta \vdash e : A$ is subscripted with a clock-context Δ containing all introduced clock variables. As such, the rule states that applying the tick constant to a suspended computation on clock κ is only well-typed if the context is well-typed *without* κ; that is, a context where κ is not free. Such a context is called κ-stable. The effect is that the tick constant cannot be used to force a suspended computation over κ under any tick abstractions over κ, and as such a fixpoint can only be unfolded once with a tick constant.

There is to our knowledge no implementation of a language with clock variables. Although the theory is sound, it can be hard to predict how they integrate in a practical setting: do they interfere with type inference? Can they be used ergonomically? What are their implementation strategies? We seek to explore this space by implementing an embedded language in Haskell designed for practical programming. Its theory is based on the simply-typed fragment of CloTT.

1.3 Our contributions

- We present CloPL, a language for co-productive programming based on Bahr et al. [2]. The semantics is given by a coinductively defined big-step operational semantics. The type system ensures that all CloPL programs are productive and thus have a well-defined semantics.
- We derive a higher-rank type inference algorithm for CloPL by adapting the work of Dunfield & Krishnaswami [6] to a guarded recursive setting.
- We implement CloPL as an embedded language in Haskell using quasi-quotation.
- In addition to a direct implementation of the operational semantics, we provide an efficient implementation of CloPL by compiling it into Haskell.
- We present some examples illustrating the practical use of CloPL.

Our implementation of CloPL, along with example programs, is available at https://github.com/adamschoenemann/clofrp.

2 CloPL

CloPL (Clocked Programming Language) is based on the theoretical foundation of the simply-typed fragment of CloTT. While CloPL is a research language it aims to explore clock quantification and guarded recursion in a practical setting,

[2]The rule is not the actual rule for tick constant application (which is more complicated), as it is not closed under substitution or weakening, but it is admissible.

Types	A, B, F	$::=$	$X \mid A \to B \mid \mathsf{Fix}\, X.\, F \mid \triangleright^\kappa A$
Terms	e	$::=$	$x \mid \lambda(x : A).e \mid \lambda(\alpha : \kappa).\, e \mid$ $e_1\, e_2 \mid e\,[\alpha] \mid e\,[\diamond] \mid e\,[\kappa] \mid \mathsf{fix} \mid$ $\diamond \mid \mathsf{fold} \mid \mathsf{unfold} \mid \mathsf{primRec}_F$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \alpha : \kappa$
Clock Contexts	Δ	$::=$	$\cdot \mid \Delta, \kappa$

Figure 1. Selected grammar rules from simply-typed CloTT.

so it aims to include many modern programming language features such as parametric polymorphism, type inference, user-defined algebraic data-types, pattern-matching, higher-rank and higher-kinded types, type-synonyms and code deriving. It does *not* include general recursion, as that would defeat the purpose of guarded recursion; instead, it provides a primitive recursion combinator and a guarded fixpoint combinator.

2.1 Type system

Figures 1 and 2 show selected syntax[3] and typing rules for the core calculus of CloPL, which is based heavily on the simply-typed fragment of CloTT. The syntax is entirely standard except for tick abstraction $\lambda(\alpha : \kappa).\, e$, tick application $e\,[\alpha]$, and clock application $e\,[\kappa]$. Contexts come in two kinds: ordinary typing contexts Γ, which can both contain assertions about the type of names, but also the clocks of ticks; and clock-contexts Δ which are simply lists of clock variables. The $\triangleright\,(\alpha : \kappa).\, A$ term from CloTT is gone, since types can no longer depend on values, and therefore they cannot depend on ticks either, and replaced with the "ordinary" clock-indexed guarded type-former $\triangleright^\kappa A$.

Recursive types are modeled using the Fix type constructor, along with the fold and unfold combinators which are similar to the traditional treatment of isorecursive types. 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 e to have type $\triangleright^\kappa A \to A$. The formation rules for recursive types are predicated on the well-formedness of $\mathsf{Fix}\, X.\, A$; namely that X can only occur strictly positively *unless* it is guarded by a $\triangleright^\kappa$ [9].

The rules in Figure 2 show that we type clock quantification and clock application in much the same way as ordinary type quantification in e.g. System F. TickAbs describes how the *later*-type is introduced through tick abstraction, while TickApp shows the elimination form. The main difference is the guarded fixpoint Fix which is indexed over a clock κ, and the tick constant application rule TConstApp whose second premise $\Gamma \vdash_\Delta$ expresses that Γ should be κ-stable by asserting that it is well-formed in a clock-context without κ.

In this calculus we would represent natural numbers as $\mathsf{Nat} := \mathsf{Fix}\, X.\, 1 + X$ and guarded streams as $\mathsf{Str}^\kappa :=$ $\mathsf{Fix}\, X.\, \mathsf{Nat} \times \triangleright^\kappa X$, giving rise to definitions:

$$
\begin{array}{rcl}
\mathsf{cons}^\kappa & : & \mathsf{Nat} \to \triangleright^\kappa \mathsf{Str}^\kappa \to \mathsf{Str}^\kappa \\
\mathsf{cons}^\kappa & := & \lambda(x : \mathsf{Nat}).\, \lambda(xs : \triangleright^\kappa \mathsf{Str}^\kappa).\, \mathsf{fold}\ \langle x, xs \rangle \\
\mathsf{hd}^\kappa & : & \mathsf{Str}^\kappa \to \mathsf{Nat} \\
\mathsf{hd}^\kappa & := & \lambda(x : \mathsf{Str}^\kappa).\, \mathsf{fst}\ (\mathsf{unfold}\ x) \\
\mathsf{tl}^\kappa & : & \mathsf{Str}^\kappa \to \triangleright^\kappa \mathsf{Str}^\kappa \\
\mathsf{tl}^\kappa & := & \lambda(xs : \mathsf{Str}^\kappa).\, \mathsf{snd}\ (\mathsf{unfold}\ xs) \\
\mathsf{const}^\kappa & := & \lambda(n : \mathsf{Nat}).\, \mathsf{fix}^\kappa (\lambda(g : \triangleright^\kappa \mathsf{Str}^\kappa).\, \mathsf{cons}^\kappa\ n\ g)
\end{array}
$$

We can then use universal quantification over clocks to model coinductive types from guarded recursive types as per Atkey & McBride [1] to define the type of coinductive streams $\mathsf{Str} := \forall\kappa.\, \mathsf{Str}^\kappa$ and then lift const^κ to produce coinductive streams as $\mathsf{const} := \lambda(n : \mathsf{Nat}).\, \Lambda\kappa.\, \mathsf{const}^\kappa\ n$. We can then lift hd^κ and tl^κ in the same way. Here, we work in a context with a *clock constant* κ_0 which is used to instantiate a clock quantified value, thus safely removing the guardedness. We then use primitive recursion to define the nth function on streams:

$$
\begin{array}{rcl}
\mathsf{hd} & : & \mathsf{Str} \to \mathsf{Nat} \\
\mathsf{hd} & := & \lambda(xs : \mathsf{Str}).\, \mathsf{hd}^{\kappa_0}\ (xs\ [\kappa_0]) \\
\mathsf{tl} & : & \mathsf{Str} \to \mathsf{Str} \\
\mathsf{tl} & := & \lambda(xs : \mathsf{Str}).\, \Lambda\kappa.\, \mathsf{tl}^\kappa\ (xs\ [\kappa])\ [\diamond] \\
\mathsf{nth} & : & \mathsf{Nat} \to \mathsf{Str} \to \mathsf{Nat} \\
\mathsf{nth} & := & \mathsf{primRec}_{1+X}\ (\\
 & & \quad \lambda(n : \mathsf{Nat}).\, \lambda(xs : \mathsf{Str}). \\
 & & \quad\quad \mathsf{case}\ \mathsf{unfold}\ n\ \mathsf{of} \\
 & & \quad\quad\quad \mathsf{inl}\ \langle\rangle \longrightarrow \mathsf{hd}\, xs \\
 & & \quad\quad\quad \mathsf{inr}\ \langle n', rec \rangle \longrightarrow rec\ (\mathsf{tl}\ xs) \\
 & &)
\end{array}
$$

The calculus we have now presented illustrates the core mechanism of clock variables which is central to CloPL, but it is not feasible to write practical programs with it alone. To do this, we shall augment the calculus with higher-rank polymorphism, user-defined algebraic data-types, higher-kinded types, pattern-matching and a bidirectional inference system.

3 Type inference

Our type inference leans heavily on that presented by Dunfield & Krishnaswami [6], which is a bidirectional inference system for higher-rank predicative polymorphism, with arbitrarily-deep quantifiers, which is (relatively) easy to implement and specify, and retains the η-law (namely that if $(\lambda x.\ f\, x) : A$ and $x \notin FV(f)$, then $f : A$). Dunfield & Krishnaswami present their inference system in two forms: A declarative system, that employs an oracle but which is simple and easy to understand; and a an algorithmic system, which directly yields an algorithm, but is more complicated. They prove that their algorithmic system is sound and complete wrt. their declarative system. We shall follow the same structure: first we present a declarative specification of CloPL and then we follow the same procedure as Dunfield & Krishnaswami to derive the algorithmic specification. For a thorough account of their approach, we refer the reader to their paper.

[3] Rules for e.g. sums and products are left out.

$$\textsc{Var}\ \frac{}{\Gamma, x : A, \Gamma' \vdash_\Delta x : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash_\Delta e : B}{\Gamma \vdash_\Delta \lambda(x : A).\ e : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash_\Delta e : A \to B \qquad \Gamma \vdash_\Delta s : A}{\Gamma \vdash_\Delta e\, s : B}$$

$$\textsc{TickAbs}\ \frac{\Gamma, \alpha : \kappa \vdash_\Delta e : A \qquad \kappa \in \Delta}{\Gamma \vdash_\Delta \lambda(\alpha : \kappa).\ e : \triangleright^\kappa A}$$

$$\textsc{ClockAbs}\ \frac{\Gamma \vdash_{\Delta,\kappa} e : A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_\Delta \Lambda\kappa.\ e : \forall\kappa.\ A} \qquad \textsc{ClockApp}\ \frac{\Gamma \vdash_\Delta e : \forall\kappa.\ A \qquad \kappa' \in \Delta}{\Gamma \vdash_\Delta e\ [\kappa'] : A\,[\kappa'/\kappa]}$$

$$\textsc{TickApp}\ \frac{\Gamma \vdash_\Delta e : \triangleright^\kappa A \qquad \Gamma' \vdash_\Delta}{\Gamma, \alpha : \kappa, \Gamma' \vdash_\Delta e\ [\alpha] : A} \qquad \textsc{TConstApp}\ \frac{\Gamma \vdash_{\Delta,\kappa} e : \triangleright^\kappa A \qquad \Gamma \vdash_\Delta}{\Gamma \vdash_{\Delta,\kappa} e\ [\diamond] : A}$$

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

$$\textsc{PrimRec}\ \frac{\Gamma \vdash_\Delta e : F[((\mathsf{Fix}\, X.\, F[X]) \times A)/X] \to A}{\Gamma \vdash_\Delta \mathsf{primRec}_F\, e : (\mathsf{Fix}\, X.\, F[X]) \to A}$$

Figure 2. Selected typing rules from simply-typed CloTT.

3.1 Declarative inference for CloPL

The declarative inference system uses a bidirectional approach with a synthesis ($\Psi \vdash e \Rightarrow A$) and a checking judgment ($\Psi \vdash e \Leftarrow A$). As in [6], we add a third *application synthesis* judgment $\Psi \vdash A \bullet e \Rrightarrow C$ that says if a (possibly polymorphic) function of type A is applied to e, then the application synthesizes type C.

3.1.1 Syntax

The syntax of CloPL can be seen in Figure 3, specified as an EBNF grammar.

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 $\forall\alpha : \chi.\ A$, however we omit the kind and write $\forall\alpha.\ A$ when we mean $\forall\alpha : \star.\ A$. 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 model clock application is just a specialized form of type application (with kind c), in the same way that we model clock abstraction using regular type abstraction. One thing that we have omitted is the term-level syntax for clock abstraction ($\Lambda\kappa.e$). Our type system will be able to infer tick abstractions from the type of terms. 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 original 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 $\mathsf{fix} : (\triangleright^\kappa A \to A) \to A$.

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

In the same way as in [1, 4], we use $\mathsf{primRec}_F$ to write functions using primitive recursion. 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 $\mathcal{C}$ applied to zero or more patterns, or n-ary tuple patterns $\langle \rho_1, \ldots, \rho_n \rangle$.

Declarative contexts are lists 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 $x :_= A$. This is important when we judge a context as κ-stable, since let-bound variables are ignored, as they are effectively just substitutions and thus cannot break productivity. Constructors are bound

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

Figure 3. The syntax of the declarative specification of CloPL.

to types as $\mathcal{C} : A$; we use such bindings to both type pattern-matches and constructor-applications. Declared types are associated with a kind as $\mathcal{T} : \chi$, while type variables are associated as $\alpha : \chi$. Finally, we assert that a type F of kind $\star \to \star$ is functorial as $\mathit{Functor}(F)$.

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.

3.1.2 Kinds of types

Types in CloPL have kinds, and these kinds can be derived inductively over the structure of a type. The derivation rules are quite standard; we show just two here, but the complete rules can be found in Appendix A. The KIND$\triangleright^{\kappa}$ rule demonstrates how to derive the kind of a later type, which requires the superscript κ to have the kind of clocks c:

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

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 [2].

3.1.3 Subtyping

As in [6], a type A is a subtype of B if $A \equiv B$ or a generalization of B. For example, $A \leq A$ and $\forall \alpha.A \leq B$ if we can instantiate α with a monotype τ such that $[\tau/\alpha]A \leq B$ (rule $\leq\forall$L). The subtyping rules of CloPL are pretty much identical to the declarative subtyping rules of [6], except extended to work with the larger source language. Rule $\leq\triangleright^{\kappa}$ derives a subtyping relation for the *later* type constructor simply by proceeding covariantly. Type application $F\,A$ also proceeds covariantly, while the function space is contravariant, as expected.

$$
\leq\triangleright^{\kappa}\ \frac{\Psi \vdash \kappa_1 \leq \kappa_2 \qquad \Psi \vdash A \leq B}{\Psi \vdash \triangleright^{\kappa_1} A \leq \triangleright^{\kappa_2} B} \qquad \leq\forall\mathrm{L}\ \frac{\Psi \vdash \tau \Rightarrow \chi \qquad \Psi \vdash [\tau/\alpha]A \leq B}{\Psi \vdash \forall \alpha{:}\chi.\, A \leq B}
$$

The complete subtyping rules can be found in Appendix B.

3.1.4 Declarative inference rules

The inference rules for CloPL (Figure 4 shows selected rules, see Appendix C for the complete picture) again closely resemble those of [6]. To accomodate our larger language, we also introduce two new judgments: $(\Psi \vdash \rho \mathrel{\not\Rightarrow} A \dashv \Psi')$ checks a pattern ρ against a type A and returns an *output context* Ψ' which extends Ψ with all bindings in ρ; $(\Psi \vdash (\rho : A \longrightarrow e) \mathrel{\Leftarrow\!\!\!\cdot} B)$ checks a branch of a case expression with pattern ρ of type A and body e against a type B.

The VAR rule simply looks up the type of a variable in the context. We simply write $x : A$ to say that we do not care if x is bound by a lambda or a let-binding.

Rule CONSTR$\Rightarrow$ 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 $\mathit{stable}_k(\Psi)$ premise serves the role of the κ-*stable* judgment in [2], which asserts that Ψ does not have κ free (i.e. there are no $x :_{\lambda} A$ bindings in Ψ where $\kappa \in fv(A)$). This is why we discriminate between let-bound and lambda-bound variables, as introducing a let-binding should not de-stabilize a context, since it just models a substitution.

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 our sytem. As Haskell has demonstrated, explicit type application is in fact quite useful, and interestingly, it also allows the programmer to express some

$\boxed{\Psi \vdash e \Leftarrow A}$ 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

$$\textsc{Var}\ \frac{(x : A) \in \Psi}{\Psi \vdash x \Rightarrow A} \qquad \textsc{Constr}\Rightarrow\ \frac{(\mathcal{C} : A) \in \Psi}{\Psi \vdash \mathcal{C} \Rightarrow A} \qquad \textsc{Sub}\ \frac{\Psi \vdash e \Rightarrow A \quad \Psi \vdash A \leq B}{\Psi \vdash e \Leftarrow B} \qquad \textsc{Anno}\ \frac{\Psi \vdash A \Mapsto \star \quad \Psi \vdash e \Leftarrow A}{\Psi \vdash (e : A) \Rightarrow A} \qquad \forall\textsc{I}\ \frac{\Psi, \alpha{:}\chi \vdash e \Leftarrow A}{\Psi \vdash e \Leftarrow \forall \alpha{:}\chi.\ A} \qquad \rightarrow\textsc{I}\ \frac{\Psi, x :_\lambda A \vdash e \Leftarrow B}{\Psi \vdash \lambda x.\, e \Leftarrow A \rightarrow B}$$

$$\forall\textsc{App}\ \frac{\Psi \vdash \tau \Mapsto \chi \quad \Psi \vdash [\tau/\alpha]A \bullet e \Rrightarrow C}{\Psi \vdash (\forall \alpha{:}\chi.\ A) \bullet e \Rrightarrow C} \qquad \rightarrow\textsc{I}\Rightarrow\ \frac{\Psi \vdash \sigma \rightarrow \tau \Mapsto \star \quad \Psi, x :_\lambda \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\, e \Rightarrow \sigma \rightarrow \tau} \qquad \rightarrow\textsc{E}\ \frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\ e_2 \Rightarrow C}$$

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

$$\triangleright^\kappa\textsc{App}\diamond\ \frac{\Psi \vdash \kappa \Mapsto \mathsf{c} \quad \mathit{stable}_\kappa(\Psi)}{\Psi \vdash (\triangleright^\kappa A) \bullet \diamond \Rrightarrow A} \qquad \textsc{TApp}\ \frac{\Psi \vdash e \Rightarrow \forall(\alpha : \chi).\ B \quad \Psi \vdash A \Mapsto \chi}{\Psi \vdash e\,\{A\} \Rightarrow [A/\alpha]B} \qquad \textsc{PrimRec}\ \frac{\mathit{Functor}(F) \in \Psi}{\Psi \vdash \mathsf{primRec}_F \Rightarrow \forall \alpha.\ F\,\langle \mathsf{Fix}\,F, \alpha\rangle \rightarrow \mathsf{Fix}\,F \rightarrow \alpha}$$

Figure 4. Selected declarative inference rules for CloPL.

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 [13]). The inference system is predicative though, so impredicative types are not first-class citizens in CloPL.

PrimRec 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. In the same vein, we model fold, unfold, fix and fmap_F as first-class polymorphic primitives, but 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 -> b) -> f a -> 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.

Pattern-matching, let-bindings and case-expressions are left out of this exposition, as they are not particularly novel; the interested reader can find their rules and a brief explanation of them in Appendix C.

This concludes the section on the declarative specification of CloPL. While the declarative rules describe the language adequately, they do not directly yield an algorithm for checking programs written in CloPL, because they enjoy guessing far too much. Specifically, the inference rules $\rightarrow$I$\Rightarrow$ and $\forall$App along with subtyping rule $\leq\forall$L require an oracle.

3.2 Algorithmic inference for CloPL

To describe an algorithm for type checking CloPL programs, we shall employ the same method as [6], 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 [6] 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.

3.2.1 Syntax

The syntax of the algorithmic system is nearly identical to the declarative system. Unlike in the declarative specification, we shall choose to split the context in two: a *global* and a *local ordered* 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.

Types are the same, but now also contain *existential type-variables* $\hat{\alpha}$. These variables represent a "hole" where we can later substitute a type, but they are not quite unification variables. They are organized in the *local context*, in which their position controls the scope of existentials and the free variables that are allowed in their solutions.

The local context Γ can contain existential variables $\hat{\alpha} : \chi$, monotype solutions to existentials $\hat{\alpha} : \chi = \tau$, and *markers* $\blacktriangleright_{\hat{\alpha}}$. Markers have no special meaning except to mark scopes in cases where we cannot mark the scope clearly using other methods. Bound names are still distinguished by 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 5.

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 makes the pattern-matching specification (and implementation) slightly more elegant.

$$\begin{array}{llll} \text{Global contexts} & \Phi & ::= & \cdot \mid \Gamma, \mathcal{C} : A \mid \Gamma, \mathcal{T} : \chi \mid \Gamma, \mathcal{P} : \forall \vec{\hat{\alpha}}.\ \hat{B_1} \to \cdots \to \hat{B_n} \to \hat{\tau} \mid Functor(F) \\ \text{Ordered contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, x :_\lambda A \mid \Gamma, x :_= A \mid \Gamma, \alpha : \chi \mid \Gamma, \hat{\alpha} : \chi \mid \Gamma, \hat{\alpha} : \chi = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}} \end{array}$$

Figure 5. Syntax for algorithmic contexts.

Precisely as in [6], local contexts are only well-formed if every context-element only depends on type variables that are located before itself in the context. The context $[\hat{\alpha}, \hat{\beta} = \hat{\alpha} \to \langle\rangle]$ is well-formed, but $[\hat{\beta} = \hat{\alpha} \to \langle\rangle, \hat{\alpha}]$ is *not*. This notion is formalized in Appendix G. Note that we write simply $x : A$ when we do not care if x was bound by a lambda or a pattern. *Context application* $[\Gamma]A$ replaces all existentials in A with their solutions in Γ, if such solutions exist (formalized in Appendix E). *Context holes* $\Gamma[\hat{\alpha} : \chi]$ means that $\hat{\alpha} : \chi$ exists in Γ, and similarly $\Gamma[\hat{\alpha} : \chi = \tau]$ yields a new a context with $\hat{\alpha}$ solved to τ without changing its order in Γ.

3.2.2 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 in the obvious way. The kind inference judgment can be seen in Appendix F.

3.2.3 Subtyping

As in [6] the algorithmic subtyping relation follows the same general pattern as the declarative one. However, it now returns an *output context* Δ, giving it the general form $\Gamma \vdash A <: B \dashv \Delta$, which says that under context Γ, type A is a subtype of B, yielding output context Δ. The two subtyping rules we showed earlier from the declarative system are shown in their algorithmic form below, along with a new rule.

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

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

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

Rule $<:\triangleright^{\kappa}$ shows a general invariant: We always make sure to fully-apply types under intermediate contexts in premises, so their arguments do not contain any existentials that are solved in their input context.

Rule $<:\forall$L is the algorithmic version of the oracular declarative rule $\leq\forall$L. Instead of guessing a type τ, we introduce an existential $\hat{\alpha} : \chi$ and substitute it for the universal variable in A. After the rule, anything introduced after the quantifier goes out of scope, so we drop it from the output context. Here, we use the marker $\blacktriangleright_{\hat{\alpha}}$ to mark the scope of the quantified type; since the premise may *solve* $\hat{\alpha}$ to $\hat{\alpha} : \chi = \tau$, or *articulate* $\hat{\alpha}$ (explained later) and thus insert new existentials *before* $\hat{\alpha}$, the marker is necessary to ensure the scope is properly maintained.

Rule $<:$InstantiateL switches to the *left-instantiation* judgment $\Gamma \vdash \hat{\alpha} : \chi :\overset{\leq}{=} A \dashv \Delta$ which yields an output context where $\hat{\alpha}$ is instantiated to a subtype of A. The first premise is the classic "occurs-check" that prevents infinite types. There is a dual *right-instantiation* judgment that assigns a supertype instead. The complete rules for both judgments can be found in Appendix H and I.

3.2.4 Instantiation

The instantiation rules are mostly similar to the instantiation rules found in [6], but extended to work with the larger source language. A few selected rules are shown below.

$$\frac{\Gamma \vdash \tau \Rightarrowtriangle \chi}{\Gamma[\hat{\alpha} : \chi] \vdash \hat{\alpha} : \chi :\overset{\leq}{=} \tau \dashv \Gamma[\hat{\alpha} : \chi = \tau]} \textsc{InstLSolve}$$

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

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

Rule InstLSolve show a base case in the judgment. When assigning an existential $\hat{\alpha} : \chi$ to a monotype τ, we simply return a context with $\hat{\alpha} : \chi$ solved to τ (after first checking that the kinds are equal).

InstLAllR demonstrates how we deal with assigning existentials to polymorphic types; our inference system relies on predicativity, so we cannot simply instantiate $\hat{\alpha} : \chi$ to a polymorphic type. Instead, we add the universal variable $\beta : \chi$ to the context and attempt to assign $\hat{\alpha} : \chi$ to B. After the assignment, β and everything to the right of it goes out of scope, so it is dropped from the output context.

InstL$\triangleright^{\kappa}$ assigns an existential to a later type by *articulating* it first. $\hat{\alpha}$ is split up into $\hat{\alpha}_1$ and $\hat{\alpha}_2$ and $\hat{\alpha}$ is assigned the type $\triangleright^{\hat{\alpha}_1}\hat{\alpha}_2$. $\hat{\alpha}_1$ and $\hat{\alpha}_2$ are inserted in the input context to the first premise *just before* $\hat{\alpha}$, so that it may refer back to them. Any markers before $\hat{\alpha}$ will also be before the articulated existentials, so correct scope is maintained. All the instantiation rules for composite types follow the same strategy of articulating the existential.

3.2.5 Algorithmic inference rules

Finally, we present selected algorithmic inference rules in Figure 6.

The three judgments correspond to their declarative counterparts, but they now yield an output context Δ. There are also algorithmic rules for branches and pattern matching, which can be found in Appendix D.

Most of the rules are simply their declarative equivalents, except that they propagate the output context of their premises.

The two oracular rules $\forall$App and $\to$I$\Rightarrow$ no longer employ guessing, but instead introduce existentials.

The subsumption rule Sub switches to subtyping which in turn might employ the the instantiation judgments to solve existentials in its output context.

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. This concludes the specification of the type inference algorithm used by CloPL.

Listing 1 shows how to encode the `map` and `maap` functions we saw previously in runnable and type checked CloPL. The concrete syntax is primarily an "ASCII-fication" of the abstract syntax, however tick abstractions $\gamma(x : \kappa).\, e$ are written with a double-backslash instead of a γ. CloPL is not whitespace sensitive, and so we terminate declarations with a full stop. As in Haskell, we "hide" higher-rank types behind a type (`CoStr` in this case) to gives us more fine-grained control over type abstraction and application. We can see that quite a few types are inferred as we only have to annotate the top-level definitions.

```
data StrF (k : Clock) a f = Cons a (|>k f) deriving Functor.
type Str (k : Clock) a = Fix (StrF k a).
data CoStr a = Cos (forall (k : Clock). Str k a).

cons : forall (k : Clock) a. a -> |>k (Str k a) -> Str k a.
cons = \x xs -> fold (Cons x xs).

uncos : forall (k : Clock) a. CoStream a -> Stream k a.
uncos = \s -> let Cos s' = s in s' {k}.

hd : forall a. CoStr a -> a.
hd = \s -> let Cons x s' = unfold (uncos {K0} s) in x.

tl : forall a. CoStr a -> CoStr a.
tl = \s -> Cos (let Cons x s' = unfold (uncos s) in s' [<>]).

map : forall (k : Clock) a b. (a -> b) -> Str k a -> Str k b.
map = \f -> fix (\g xs ->
  let Cons x xs' = unfold xs in
  let ys = \\(af : k) -> g [af] (xs' [af]) in
  cons (f x) ys
).

maapk : forall (k : Clock) a b.
  (a -> b) -> CoStr a -> Str k b.
maapk = \f ->
  fix (\g xs ->
    let fhd = f (hd xs) in
    let ftltl = \\(af : k) -> g [af] (tl (tl xs)) in
    let ftl = \\(af : k) -> cons (f (hd (tl xs))) ftltl in
    cons fhd ftl
  ).

maap : forall a b. (a -> b) -> CoStr a -> CoStr b.
maap = \f xs -> Cos (maapk f xs).
```

Listing 1. `map` and `maap` from [1] encoded in CloPL

4 Operational semantics of CloPL

The operational semantics of CloPL are derived from the reduction semantics specified for CloTT in [2].

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 [7], 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[4].

4.1 Values and contexts

Values represent normal forms of the terms in CloPL. Figure 7 describes the syntax of values and evaluation contexts.

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.

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

4.2 One-step semantics

The one-step semantics can be seen in Figure 8. The main relation $\Xi \vdash e \Downarrow v$ evaluates a CloPL expression e to a value v under an evaluation context Ξ. The pattern-matching semantics $\Xi \vdash \rho \lfloor v \dashv \Xi'$ 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.

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ 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

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma} \qquad \textsc{Constr}\Rightarrow\ \frac{(\mathcal{C} : A) \in \Phi}{\Gamma \vdash \mathcal{C} \Rightarrow A \vdash \Gamma} \qquad \textsc{Anno}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow A \dashv \Delta} \qquad \forall\text{I}\ \frac{\Gamma, \alpha : \chi \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall \alpha : \chi.\ A \dashv \Delta}$$

$$\forall\textsc{App}\ \frac{\Gamma, \hat{\alpha} : \chi \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha : \chi.\ A \bullet e \Rrightarrow C \dashv \Delta} \qquad \rightarrow\text{I}\ \frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \rightarrow B \dashv \Delta} \qquad \textsc{Sub}\ \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}$$

$$\rightarrow\text{I}\Rightarrow\ \frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat{\alpha} \rightarrow \hat{\beta} \dashv \Delta} \qquad \rightarrow\text{E}\ \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\ e_2 \Rightarrow C \dashv \Delta} \qquad \triangleright^{\kappa}\textsc{App}\ \frac{\Gamma \vdash \kappa \Mapsto \mathsf{c} \qquad \Gamma \vdash x \Leftarrow \kappa \dashv \Delta}{\Gamma \vdash (\triangleright^{\kappa} A) \bullet [x] \Rrightarrow A \dashv \Delta}$$

$$\hat{\alpha}\textsc{App}\ \frac{\Gamma[\hat{\alpha}_2 : \star, \hat{\alpha}_1 : \star, \hat{\alpha} : \star = \hat{\alpha}_1 \rightarrow \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2 \dashv \Delta} \qquad \triangleright^{\kappa}\hat{\alpha}\textsc{App}\ \frac{\Gamma[\hat{\alpha}_2 : \chi, \hat{\alpha}_1 : \mathsf{c}, \hat{\alpha} : \star = \triangleright^{\hat{\alpha}_1} \hat{\alpha}_2] \vdash x \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha} : \star] \vdash \hat{\alpha} \bullet [x] \Rrightarrow \hat{\alpha}_2 \dashv \Delta} \qquad \triangleright^{\kappa}\textsc{App}\diamond\ \frac{\Gamma \vdash \kappa \Mapsto \mathsf{c} \qquad \mathit{stable}_{\kappa}(\Gamma)}{\Gamma \vdash (\triangleright^{\kappa} A) \bullet [\diamond] \Rrightarrow A \dashv \Gamma}$$

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

Figure 6. Selected algorithmic inference rules

Values $v ::= \diamond \mid \lambda_{\Xi}\, x.\ e \mid \gamma_{\Xi}\, \alpha.\ e \mid \langle v_1, \ldots, v_n \rangle \mid \mathcal{C}\, v_1, \ldots, v_n \mid \mathsf{fold}\, v$

Evaluation contexts $\Xi ::= \cdot \mid \Xi, x \mapsto v, \mathit{fmap}(F) \mapsto v$

Figure 7. Values and evaluation contexts in CloPL.

Most of the rules are quite straight-forward. Tick says that the tick constant simply evaluates back to itself. 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. 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 Bahr et al. [2], 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 $\mathsf{fix}\, f$ 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 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.

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

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

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

$$\textsc{Fmap}\ \frac{(fmap(F) \mapsto v) \in \Xi}{\Xi \vdash \mathsf{fmap}_F \Downarrow v} \qquad \textsc{Ann}\ \frac{\Xi \vdash e \Downarrow v}{\Xi \vdash e : A \Downarrow v}$$

$$\textsc{TypeApp}\ \frac{\Xi \vdash e \Downarrow v}{\Xi \vdash e\,\{A\} \Downarrow v} \qquad \textsc{Fold}\ \frac{\Xi \vdash e \Downarrow v}{\Xi \vdash \mathsf{fold}\, e \Downarrow \mathsf{fold}\, v} \qquad \textsc{Unfold}\ \frac{\Xi \vdash e \Downarrow \mathsf{fold}\, v}{\Xi \vdash \mathsf{unfold}\, e \Downarrow v}$$

$$\textsc{Tuple}\ \frac{\Xi \vdash e_1 \Downarrow v_1 \ \ldots\ \Xi \vdash e_n \Downarrow v_n}{\Xi \vdash \langle e_1, \ldots, e_n \rangle \Downarrow \langle v_1, \ldots, v_n \rangle} \qquad \textsc{Fix}\ \frac{\Xi \vdash \lambda f.\ f\ (\gamma(\_ : \kappa).\ \mathsf{fix}\, f) \Downarrow v}{\Xi \vdash \mathsf{fix} \Downarrow v}$$

$$\textsc{Let}\ \frac{\Xi \vdash e' \Downarrow v' \qquad \Xi \vdash \rho \lfloor v' \dashv \Xi' \qquad \Xi' \vdash e \Downarrow v}{\Xi \vdash \mathsf{let}\ \rho = e'\ \mathsf{in}\ e \Downarrow v} \qquad \textsc{Case}\ \frac{\Xi \vdash e \Downarrow v \qquad \text{Smallest } i \text{ such that } \Xi \vdash \rho_i \lfloor v \dashv \Xi' \qquad \Xi' \vdash e_i \Downarrow v_i}{\Xi \vdash \mathsf{case}\ e\ \mathsf{of}\ (\rho_1 \longrightarrow e_1, \ldots, \rho_n \longrightarrow e_n) \Downarrow v_i}$$

$$\textsc{PrimRec}\ \frac{\Xi \vdash \lambda b\, o.\ b\,(\mathsf{fmap}_F\ (\lambda i.\ \langle i, \mathsf{primRec}_F\ b\ i\rangle)\ (\mathsf{unfold}\ o)) \Downarrow v}{\Xi \vdash \mathsf{primRec}_F \Downarrow v} \qquad \textsc{LamApp}\ \frac{\Xi \vdash e_1 \Downarrow \lambda_{\Xi'}\, x.\ e \quad \Xi \vdash e_2 \Downarrow v_2 \quad \Xi', x \mapsto v_2 \vdash e \Downarrow v}{\Xi \vdash e_1\, e_2 \Downarrow v}$$

$$\textsc{TickApp}\ \frac{\Xi \vdash e_1 \Downarrow \gamma_{\Xi'}\, x.\ e \quad \Xi \vdash e_2 \Downarrow v_2 \quad \Xi', x \mapsto v_2 \vdash e \Downarrow v}{\Xi \vdash e_1\,[e_2] \Downarrow v} \qquad \textsc{ConstrApp}\ \frac{\Xi \vdash e_1 \Downarrow (\mathcal{C}\, v_1, \ldots, v_n) \qquad \Xi \vdash e_2 \Downarrow v}{\Xi \vdash e_1\, e_2 \Downarrow (\mathcal{C}\, v_1, \ldots, v_n, v)}$$

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

Figure 8. Big-step operational one-step-semantics of CloPL.

4.3 Coinductive tick-semantics

The one-step semantics given in 4.2 will only evaluate a CloPL 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 9.

The tick-semantics is quite simple. $\Xi \vdash v \ggg v'$ just proceeds structurally over values. If it encounters primitives, it just returns them ($\ggg$Value). When it goes under a constructor ($\ggg$Constr), or a tuple ($\ggg$Tuple), it attempts to force any delayed expressions under them. Since we only have guarded recursion to worry about at this point, this is safe. Values fold must have constructors v directly under them, so we simply proceed into v (rule $\ggg$Fold). Rule $\lightning$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 $\lightning$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 CloPL, which all terms should be if they typecheck.

Finally, to evaluate a program e in CloPL, 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.

We have implemented a simple evaluator derived from the operational semantics in Haskell; the implementation is described in our master thesis [10].

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

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

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

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

$$\frac{\Xi' \vdash v \ggg v'}{\Xi \vdash \mathsf{fold}\, v \ggg \mathsf{fold}\, v'}\ \ggg\textsc{Fold} \qquad \frac{}{\Xi \vdash v \ggg v}\ \ggg\textsc{Value}$$

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

Figure 9. Tick-semantics for CloPL.

$$
\begin{array}{rcl}
[\![\mathcal{C}]\!] & = & \mathcal{C} \\
[\![x]\!] & = & x \\
[\![e : t]\!] & = & e :: t \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\
[\![\lambda(x : t).\, e]\!] & = & \lambda(!x :: [\![t]\!]).\, [\![e]\!] \\
[\![\gamma(x : k).\, e]\!] & = & \lambda!x.\, [\![e]\!] \\
[\![(e_1, ..., e_n)]\!] & = & ([\![e_1]\!], ..., [\![e_n]\!]) \\
[\![\mathsf{let}\ \rho = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{let}\ [\![p]\!] = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\
[\![\mathsf{case}\ e\ \mathsf{of}\ cs]\!] & = & \mathsf{case}\ [\![e]\!]\ \mathsf{of}\ [\![cs]\!] \\
[\![e\,\{t\}]\!] & = & [\![e]\!]\ @[\![t]\!] \\
[\![\mathsf{fmap}_F]\!] & = & \mathsf{fmap} \\
[\![\mathsf{primRec}_F]\!] & = & \mathsf{primRec} \\
[\![\mathsf{fix}]\!] & = & \mathsf{gfix}
\end{array}
$$

Figure 10. Compilation of CloPL terms to Haskell terms.

5 Compiling to Haskell

While we have implemented the operational semantics as an evaluator in Haskell, its runtime characteristics leave much to be desired. Although plenty of opportunities for optimizations exist, it is a much easier path to performance to simply compile CloPL programs to Haskell, and thus exploit the many optimizations that Haskell has accrued over the years. We have to preserve the semantics we have given above in the compiled code. To do so, we compile data-declarations to their Haskell equivalents, but use `BangPatterns` to ensure that the constructors are strict in their arguments. Similarly, all pattern-matches and let-bindings are decorated with bangs to make them strict. To ensure this strictness does not introduce divergence, the later type $\triangleright^\kappa A$ is compiled to a *thunk* as $[\![\triangleright^\kappa A]\!] = 1 \to A$, and the tick constant simply becomes $\langle\rangle$. We define our primitive types and combinators in Haskell:

```
data ClockKind = K0 -- lifted with DataKinds

data Fix f = Fold (f (Fix f))

gfix :: ((() -> a) -> a) -> a
gfix f = let x = f (\_ -> x)
         in  x

unfold :: Fix f -> f (Fix f)
unfold (Fold f) = f

primRec :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
primRec fn (Fold f) =
  fn (fmap (\y -> (y, primRec fn y)) f)
```

The guarded fixpoint `gfix` is interpreted simply as the normal fixpoint, but instead of a normal endofunction it takes a guarded function, which is just a thunk in our interpretation.

Figures 10 and 11 show how we interpret CloPL terms and types in Haskell – it is entirely straightforward, and the entire compiler is no more than 170 lines including whitespace and a few comments. The product is a `QuasiQuoter` that one can use to write CloPL programs inside Haskell while having them type checked at compile-time.

$$
\begin{array}{rcl}
[\![\mathcal{T}]\!] & = & \mathcal{T} \\
[\![x]\!] & = & x \\
[\![t_1\ t_2]\!] & = & [\![t_1]\!]\ [\![t_2]\!] \\
[\![t_1 \to t_2]\!] & = & [\![t_1]\!] \to [\![t_2]\!] \\
[\![\forall(\alpha : \chi).\, t]\!] & = & \mathsf{forall}\ (\alpha :: [\![\chi]\!]).\, [\![t]\!] \\
[\![\mathsf{Fix}\ t]\!] & = & \mathsf{Fix}\ [\![t]\!] \\
[\![\langle t_1, ..., t_n \rangle]\!] & = & ([\![t_1]\!], ..., [\![t_n]\!]) \\
[\![\triangleright^\kappa t]\!] & = & () \to [\![t]\!]
\end{array}
$$

Figure 11. Compilation of CloPL types to Haskell types.

6 Example CloPL programs

Now that we have presented CloPL, let us look at a few example programs that demonstrate the language. A classical corecursive definition is the infinite stream of natural numbers, which we can express in CloPL as:

```
conats : CoStr Int.
conats = Cos (fix (\g ->
  fold (Cons 0 (\\(af : k) -> map (\x -> x + 1) (g [af])))
)).
```

By defining inductive natural numbers, we can implement the `nth` function from Section 2:

```
nth : Nat -> CoStr Nat -> Nat.
nth = \n xs ->
  let Cos xs' = xs in
  let fn = \n xs' ->
    case n of
    | Z -> hdk xs'
    | S (n', r) -> r (tlk xs' [<>]) in
  primRec {NatF} fn n xs'.
```

Atkey & McBride [1] equip their later modality with an applicative structure. We can define this structure ourselves instead of admitting them as primitives.

```
pure : forall (k : Clock) a. a -> |>k a.
pure = \x -> \\(af : k) -> x.

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.

-- ("delayed map" instead of fmap since fmap is reserved)
dmap : forall (k : Clock) a b. (a -> b) -> |>k a -> |>k b.
dmap = \f la -> app (pure f) la.
```

As Atkey & McBride [1] also mention, clock variables can be used to implement and reason about perculiar circular programs that lazy evaluation permits. Their main example is the `replaceMin` function that replaces all values in a binary tree with the minimum value in the tree *in only one pass*. Conceptually, the computation of the minimum and the construction of the new tree exist at different moments in time, in spite of both being *defined* in one pass. Such programs are very clever, but also extremely easy to get wrong which often results in non-productive programs.

To encode this in CloPL we will first need to define a *feedback* combinator, which feeds the result of a guarded recursive function back into itself.

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

Here we can leverage the expressive power that CloPL's parametric polymorphism and type constructors gives us. We then define a new type `Delay` that is a flipped version of the later type. We also define the type of binary trees.

```
data Delay a (k : Clock) = Delay (|>k a).
data TreeF a f = Leaf a | Br f f deriving Functor.
type Tree a = Fix (TreeF a).
```

Defining $\mathsf{br}\,x\,y = \mathsf{fold}\,(\mathsf{Br}\,x\,y)$ and $\mathsf{leaf}\,x = \mathsf{fold}\,(\mathsf{Leaf}\,x)$, we can then define the main computation by primitive recursion, which we will pass to `feedback`. Finally, we can instantiate the clock quantifier to remove the guardedness and then safely force the computation with the tick constant.

```
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 (dmap 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 (dmap br l') r'), m')
).
replaceMin : Tree Int -> Tree Int.
replaceMin = \t ->
  let Delay t' = feedback {K0} (replaceMinBody t)
  in t' [<>].
```

7 Related and future work

We have presented CloPL which is a language for safe coprogramming that we have implemented as an embedded language in Haskell. We have defined the core syntax and type inference rules for CloPL, along with a big-step operational semantics and a simple method for compiling CloPL programs to executable Haskell code. We have shown a few examples of real executable CloPL programs that demonstrate that clock quantification can be coupled with modern inference techniques to facilitate ergonomic coprogramming.

CloPL is based on CloTT, which is a product of a line of research [1, 3, 5] revolving around ensuring productivity by modeling coinductive types using guarded recursive types. Guarded recursion [9] has also been used for safe functional reactive programming [7, 8] which also saw an implementation aimed at web-programming. That line of work did not include clock quantification, but instead exploited guarded recursion to give a type system and operational semantics that guaranteed the absense of implicit memory leaks in higher-order FRP programs. Cave et al. [4] expanded on this and removed the guarded fixpoint combinator and instead added iteration and co-iteration operators, along with greatest- and least-fixpoint types to enable fine-grained liveness properties to be expressed in the type system. Their system can distinguish between values that *may eventually* occur and those that *must eventually* occur; a distinction we cannot make in CloPL due to the dual role of the Fix type along with the guarded fixpoint combinator. It would be interesting to explore if such a distinction could be incorporated into CloPL.

Very recently, Severi [11, 12] presented a lambda calculus with two *silent* modalities for recursion. Her calculus also models coinductive types from guarded recursive types, but uses an integer-indexed version of the guarded type-former along with a "everything now" type operator, instead of clock quantification and instantiation. The term *silent* refers to the fact that the expression language does *not* contain syntax to eliminate and introduce the modalities, which is very impressive. The downside is that the type system is very complicated, and type inference requires an exponential-time algorithm and a linear integer programming solver.

Attempting to eliminate introduction and elimination forms from CloPL could be an interesting avenue of research, although one must balance the additional complexity penality incurred in the type system against the decreased complexity of the expression language. One of CloPL's primary benefits as a practically-oriented language is that its type system and inference system has a very simple specification and implementation.

In spite of the many features of CloPL, there are still certain features lacking in order to be considered "fully-applicable" as a practical language; most critically is the lack of coverage-checking for pattern matches. We believe that a simple implementation is feasible using standard techniques such as decision trees.

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 CloPL to the full CloTT specification would be interesting.

Acknowledgments

I would like to thank my supervisors on my Master's thesis, on which this paper is based, Rasmus Møgelberg and Patrick Bahr. An extra thanks goes out to Patrick for spurring me on to write this paper and providing feedback on it.

References

[1] R. Atkey and C. McBride. 2013. Productive Coprogramming with Guarded Recursion *(ICFP '13)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/2500365.2500597

[2] P. Bahr, H.B. Grathwohl, and R.E. Møgelberg. 2017. The Clocks Are Ticking: No More Delays! Reduction semantics for type theory with guarded recursion *(LICS '17)*. ACM/IEEE.

[3] A. Bizjak, H.B. Grathwohl, R. Clouston, R.E. Møgelberg, and L. Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. *CoRR* abs/1601.01586 (2016). arXiv:1601.01586

[4] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. 2014. Fair Reactive Programming *(POPL '14)*. ACM, New York, NY, USA, 361–372. https://doi.org/10.1145/2535838.2535881

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

[6] J. Dunfield and N.R. Krishnaswami. [n. d.]. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism *(ICFP '13)*.

[7] N.R. Krishnaswami. 2013. Higher-Order Reactive Programming without Spacetime Leaks *(ICFP '13)*.

[8] N.R. Krishnaswami and N. Benton. 2011. Ultrametric semantics of reactive programs. IEEE, 257–266.

[9] H. Nakano. 2000. A modality for recursion *(LICS '00)*. IEEE, 255–266.
[10] A. Schønemann. 2018. Functional Reactive Programming in CloTT. (2018). unpublished master thesis, available online at https://lybaek.adamschoenemann.dk/f/af8692d18b944ba5ab2d/?dl=1.
[11] P. Severi. 2017. A Light Modality for Recursion. In *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 499–516.
[12] P. Severi. 2017. Two Light Modalities for Recursion. *arXiv preprint arXiv:1801.00285* (2017).
[13] J.B. Wells. 1999. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156.

Appendix A Declarative kind judgment

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

$$\textsc{KindVar}\ \frac{(\alpha : \chi) \in \Psi}{\Psi \vdash \alpha \Mapsto \chi} \qquad \textsc{KindFree}\ \frac{(\mathcal{T} : \chi) \in \Psi}{\Psi \vdash \mathcal{T} \Mapsto \chi}$$

$$\textsc{Kind}\forall\ \frac{\Psi, \alpha : \chi \vdash A \Mapsto \star}{\Psi \vdash \forall \alpha : \chi.\, A \Mapsto \star} \qquad \textsc{Kind}\to\ \frac{\Psi \vdash A \Mapsto \star \qquad \Psi \vdash B \Mapsto \star}{\Psi \vdash A \to B \Mapsto \star}$$

$$\textsc{KindTuple}\ \frac{\Psi \vdash A_1 \Mapsto \star \qquad \ldots \qquad \Psi \vdash A_n \Mapsto \star}{\Psi \vdash \langle A_1, \ldots, A_n \rangle \Mapsto \star}$$

$$\textsc{KindApp}\ \frac{\Psi \vdash F \Mapsto (\chi_1 \to \chi_2) \qquad \Psi \vdash B \Mapsto \chi_1}{\Psi \vdash F\, B \Mapsto \chi_2}$$

$$\textsc{KindFix}\ \frac{\Psi \vdash F \Mapsto (\star \to \star)}{\Psi \vdash \mathsf{Fix}\, F \Mapsto \star} \qquad \textsc{Kind}\triangleright^{\kappa}\ \frac{\Psi \vdash \kappa \Mapsto \mathsf{c} \qquad \Psi \vdash A \Mapsto \star}{\Psi \vdash \triangleright^{\kappa} A \Mapsto \star}$$

Figure A.1. Deriving kinds of types in CloPL.

Appendix B Declarative subtyping judgment

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

$$\leq\textsc{Var}\ \frac{\alpha : \chi \in \Psi}{\Psi \vdash \alpha \leq \alpha} \qquad \leq\textsc{Free}\ \frac{\mathcal{T} : \chi \in \Psi}{\Psi \vdash \mathcal{T} \leq \mathcal{T}}$$

$$\leq\!\to\ \frac{\Psi \vdash B_1 \leq A_1 \qquad \Psi \vdash A_2 \leq B_2}{\Psi \vdash A_1 \to A_2 \leq B_1 \to B_2}$$

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

$$\leq\textsc{Tuple}\ \frac{\Psi \vdash A_1 \leq B_1 \qquad \ldots \qquad \Psi \vdash A_n \leq B_n}{\Psi \vdash \langle A_1, \ldots, A_n \rangle \leq \langle B_1, \ldots, B_n \rangle}$$

$$\leq\textsc{App}\ \frac{\Psi \vdash F \leq G \qquad \Psi \vdash A \leq B}{\Psi \vdash F\, A \leq G\, B} \qquad \leq\textsc{Fix}\ \frac{\Psi \vdash F \leq G}{\Psi \vdash \mathsf{Fix}\, F \leq \mathsf{Fix}\, G}$$

$$\leq\triangleright^{\kappa}\ \frac{\Psi \vdash \kappa_1 \leq \kappa_2 \qquad \Psi \vdash A \leq B}{\Psi \vdash \triangleright^{\kappa_1} A \leq \triangleright^{\kappa_2} B}$$

Figure B.1. Subtyping in CloPL.

Appendix C Full declarative inference rules

The complete declarative inference rules are shown in Figure C. We explain the rules that are omitted from the main exposition here.

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

$\boxed{\Psi \vdash e \Leftarrow A}$ 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'

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

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

$$\frac{\Psi \vdash \sigma \to \tau \Mapsto \star \quad \Psi, x :_\lambda \sigma \vdash e \Leftarrow \tau}{\Psi \vdash \lambda x.\, e \Rightarrow \sigma \to \tau}\ \to\mathrm{I}\Rightarrow \qquad \frac{\Psi \vdash e_1 \Rightarrow A \quad \Psi \vdash A \bullet e_2 \Rrightarrow C}{\Psi \vdash e_1\, e_2 \Rightarrow C}\ \to\mathrm{E} \qquad \frac{\Psi \vdash e \Leftarrow A}{\Psi \vdash (A \to C) \bullet e \Rrightarrow C}\ \to\textsc{App}$$

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

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

$$\frac{\Psi \vdash e \Rightarrow \forall (\alpha : \chi).\ B \quad A \Mapsto \chi}{\Psi \vdash e\,\{A\} \Rightarrow [A/\alpha]B}\ \textsc{TApp} \qquad \frac{}{\Psi \vdash \mathsf{fold} \Rightarrow \forall (\alpha : \star \to \star).\ \alpha\,(\mathsf{Fix}\,\alpha) \to \mathsf{Fix}\,\alpha}\ \textsc{Fold}\Rightarrow$$

$$\frac{}{\Psi \vdash \mathsf{unfold} \Rightarrow \forall (\alpha : \star \to \star).\ \mathsf{Fix}\,\alpha \to \alpha\,(\mathsf{Fix}\,\alpha)}\ \textsc{Unfold}\Rightarrow \qquad \frac{}{\Psi \vdash \mathsf{fix} \Rightarrow \forall (\kappa : \mathsf{c}), \alpha.\ (\triangleright^\kappa \alpha \to \alpha) \to \alpha}\ \textsc{Fix}$$

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

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

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

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

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

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

Figure C.1. Full declarative inference rules for CloPL.

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'.

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 $\Psi, (x :_= [\tau/\alpha]A) \vdash e \Leftarrow B$.

Appendix D Full algorithmic inference system

See Figures D.1 and D.2.

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 counter-part, 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 CloPL 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 D.3.

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.

Appendix E Context substitution

$$
\begin{array}{lcl}
[\Gamma]\alpha & = & \alpha \\
[\Gamma]\mathsf{1} & = & \mathsf{1} \\
[\Gamma[\hat\alpha = \tau]]\hat\alpha & = & [\Gamma[\hat\alpha = \tau]]\tau \\
[\Gamma[\hat\alpha]]\hat\alpha & = & \hat\alpha \\
[\Gamma](A \to B) & = & ([\Gamma]A) \to ([\Gamma]B) \\
[\Gamma](A\,B) & = & ([\Gamma]A)\,([\Gamma]B) \\
[\Gamma](\mathsf{Fix}\ F) & = & \mathsf{Fix}\ ([\Gamma]F) \\
[\Gamma]\langle A_1, \ldots, A_n\rangle & = & \langle [\Gamma]A_1, \ldots, [\Gamma]A_n\rangle \\
[\Gamma](\forall\alpha.\ A) & = & \forall\alpha.\ ([\Gamma]A)
\end{array}
$$

Appendix F Algorithmic kind inference

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

$$\frac{(\alpha : \chi) \in \Gamma}{\Gamma \vdash \alpha \Mapsto \chi}\ \textsc{KindVar} \qquad \frac{(\mathcal{T} : \chi) \in \Phi}{\Gamma \vdash \mathcal{T} \Mapsto \chi}\ \textsc{KindFree}$$

$$\frac{\Gamma \vdash A \Mapsto \star \qquad \Gamma \vdash B \Mapsto \star}{\Gamma \vdash A \to B \Mapsto \star}\ \textsc{Kind}{\to}$$

$$\frac{\Gamma \vdash A_1 \Mapsto \star \qquad \ldots \qquad \Gamma \vdash A_n \Mapsto \star}{\Gamma \vdash \langle A_1, \ldots, A_n\rangle \Mapsto \star}\ \textsc{KindTuple}$$

$$\frac{\Gamma, \alpha : \chi \vdash A \Mapsto \star}{\Gamma \vdash \forall\alpha : \chi.\ A \Mapsto \star}\ \textsc{Kind}{\forall}$$

$$\frac{\Gamma \vdash F \Mapsto (\chi_1 \to \chi_2) \qquad \Gamma \vdash B \Mapsto \chi_1}{\Gamma \vdash F\,B \Mapsto \chi_2}\ \textsc{KindApp}$$

$$\frac{\Gamma \vdash F \Mapsto (\star \to \star)}{\Gamma \vdash \mathsf{Fix}\,F \Mapsto \star}\ \textsc{KindFix}$$

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

$$\frac{\hat\alpha : \chi = \tau \in \Gamma}{\Gamma \vdash \hat\alpha \Mapsto \chi}\ \textsc{KindEVarSolved}$$

Appendix G Well-formedness of algorithmic contexts

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

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

$$\frac{\Gamma\ ctx \qquad x \notin \mathsf{dom}\,(\Gamma) \qquad \Gamma \vdash A \Mapsto \star}{\Gamma, x : A\ ctx}\ \textsc{VarCtx}$$

$$\frac{\Gamma\ ctx \qquad \hat\alpha \notin \mathsf{dom}\,(\Gamma)}{\Gamma, \hat\alpha : \chi\ ctx}\ \textsc{EVarCtx}$$

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

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

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ 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

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

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

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

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

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

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

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

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

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

Figure D.1. Algorithmic inference rules of CloPL (part 1).

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

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

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

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

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

$$\frac{\Gamma \vdash e' \Rightarrow A \dashv \Theta \qquad \Theta, x :_{=} [\Theta]A \vdash e \Leftarrow [\Theta]B \dashv \Delta}{\Gamma \vdash \mathsf{let}\ x\ = e'\ \mathsf{in}\ e \Leftarrow B \dashv \Delta}\ \textsc{Let} \qquad \frac{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ (\rho \longrightarrow e) \Leftarrow B \vdash \Gamma \qquad \rho \neq x}{\Gamma \vdash \mathsf{let}\ \rho\ = e'\ \mathsf{in}\ e \Leftarrow B \dashv \Delta}\ \textsc{LetPat}$$

Figure D.2. Algorithmic inference rules of CloPL (part 2).

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

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

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

Figure D.3. Example derivation with pattern matching.

Appendix H Complete left-instantiation rules

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

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

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

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

$$\frac{\Gamma \vdash A_1 \Rightarrow (\chi_1 \to \chi_2) \qquad \Gamma[\hat{\alpha}_2 : \chi_1,\ \hat{\alpha}_1 : \chi_1 \to \chi_2,\ \hat{\alpha} : \chi_2 = \hat{\alpha}_1\ \hat{\alpha}_2] \vdash \hat{\alpha}_1 : \chi_1 \to \chi_2 \mathrel{:\overset{\leq}{=}} A_1 \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 : \chi_1 \mathrel{:\overset{\leq}{=}} [\Theta] A_2 \dashv \Delta}{\Gamma[\hat{\alpha} : \chi_2] \vdash \hat{\alpha} : \chi_2 \mathrel{:\overset{\leq}{=}} A_1\ A_2 \dashv \Delta}\ \textsc{InstLApp}$$

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

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

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

Figure H.1. Left-instantiation of existentials in CloPL.

Appendix I Complete right-instantiation rules

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

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

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

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

$$\frac{\Gamma \vdash A_1 \Rightarrow (\chi_1 \to \chi_2) \qquad \Gamma[\hat{\alpha}_2:\chi_1,\ \hat{\alpha}_1:\chi_1 \to \chi_2,\ \hat{\alpha}:\chi_2 = \hat{\alpha}_1\ \hat{\alpha}_2] \vdash A_1 \mathrel{\overset{\leq}{=:}} \hat{\alpha}_1 : \chi_1 \to \chi_2 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \mathrel{:\overset{\leq}{=}} \hat{\alpha}_2 : \chi_1 \dashv \Delta}{\Gamma[\hat{\alpha}:\chi_2] \vdash A_1\ A_2 \mathrel{\overset{\leq}{=:}} \hat{\alpha}:\chi_2 \dashv \Delta}\ \textsc{InstRApp}$$

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

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

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

Figure I.1. Right-instantiation of existentials in CloPL.