Implementation of Type Theory based on dependent Inductive and Coinductive Types

Florian Engel

November 20, 2020

6

7

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-
Naturwissenschaftliche
Fakultät

Programiersprachen

Masterarbeit

Implementation of Type Theory based on dependent inductive and coinductive types

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Programiersprachen
Florian Engel, `florian.engel@student.uni-tuebingen.de`, 2020

Bearbeitungszeitraum: von-bis

Betreuer/Gutachter: Prof. Dr. Klaus Ostermann, Universität Tübingen
Zweitgutachter: Prof. Dr. Max Mustermann, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Florian Engel (Matrikelnummer 3860700), November 20, 2020

Abstract

Dependent types are an useful tool to restrict types even further then types of strongly typed languages like Haskell. This gives us further type safety. With them we can also proof theorems. Coinductive types allow us to define types by their observations rather then by their constructors. This is useful for infinite types like streams. In many common dependently typed languages , like coq and agda, we can define inductive types which depend on values and coinductive types but not coinductive types, which depend on values.

In this work I will first give a survey of coinductive types in these languages and then implement the type theory from [BG16]. This type theory has both dependent inductive types and dependent coinductive types. In this type theory the dependent function space becomes definable. This leads to a more symmetrical approach of coinduction in dependently typed languages.

Contents

1. Introduction

In functional programming we have functions which get an input and produce an output. These functions don't depend on multiple values i.e. if there is no IO involved, they produce for the same input always the same output. For example, if we call a function `or` on the values `true` and `false` we always get `true`. This makes the code more predictable.

The `or` function should only be working on Booleans. To call it on strings `'foo'` and `'bar'` wouldn't make sense i.e. there is no defined output for these inputs. To prevent calls like these, some functional programming languages introduced types. Types contain only certain values. For example the type for truth values contains only the values for true and false. In Haskell we can define it like the following:

```
data Bool = True | False
```

This says we can construct values of type `Bool` with the constructors `True` and `False`. These types which have constructors are called inductive types. We can then define `or` like this.

```
or :: Bool -> Bool -> Bool
or True _    = True
or _    True = True
or _    _    = False
```

Here, we just list equations which say what the output for a given input is. For example, in the first equation, we say if the first value is constructed with the constructor `True`, we give back `True`. We don't care about the second value, therefor we write _. We are matching on the construction of the input values. Therefor we call this method pattern matching. If we call this function somewhere in the code on values which aren't of type `Bool`, Haskell won't compile our code. Instead it gives back a type error.

If we now want to change `Bool` to a three-valued logic, we have to add a third constructor to `Bool`. After that we have to change every function which pattern matches on `Bool`. If there are a lot of those kinds of functions, this would be lot of repetitive work. If Haskell would have coinductive type, this could be a lot less work. Coinductive types are types which are, in contrary to inductive types defined over their destruction. So we could define `Bool` over its destructors. These would be or, and, etc.

Through this work we will explain coinductive types at the examples of streams and functions. They will be generalized to partial streams and the Pi type in dependently typed languages. Streams are lists which are infinitely long. They are useful for modeling many IO interactions. For example, a chat of a text messenger might be infinitely long. We can never know if the chat is finished. This is of course limited by the hardware, but we are interested in abstract models. Functions are used every where in functional programming. In most of these languages they are first-class objects. But in languages with coinductive types we can define them. If we only have inductive and coinductive types, we get a symmetrical language. This is useful, because then we can change an inductive type to a coinductive one and vice versa. It is straight forward to add functions which destruct an inductive type by pattern matching on the constructor. But it is hard to add a new constructor. We then have to add this constructor to every pattern matching on that type. For coinductive types its the other way around. For more on this see [BJSO19]. In the implemented syntax we can define streams like the following:

```
codata Stream⟨A : Set⟩ : Set where Hd : Stream (succ @ k)
  Hd : Stream -> A
  Tl : Stream -> Stream
```

And functions like follows:

```
codata Fun⟨A : Set, B : Set⟩ : Set where
  Inst : (x : A) -> Fun -> B
```

We can generalize streams to partial streams as the following:

```
codata PStr⟨A : Set⟩ : (n : Conat) -> Set where
  Hd : (k : Conat) -> PStr (succ @ k) -> A
  Tl : (k : Conat) -> PStr (succ @ k) -> PStr @ k
```

And functions to the Pi type.

```
codata Pi⟨A : Set, B : (x : A) -> Set⟩ : Set where
  Inst : (x : A) -> Pi -> B @ x
```

The rest of this thesis will be structured as follows:

- In chapter 2 we will see how coinductive types can be defined. Here we will be defining the stream and function type. We will also define some functions on the stream.
- We will see in chapter 3 how coinductive types are defined in the dependently typed languages coq and agda. We will see that we can define them positive or negative. We will show why defining them positive leads to problems.
- In chapter 4 we see how they are defined in [BG16]. With this theory we can than define coinductive types which depend on values. But we can not define types which depend on types.
- We will then in chapter 5 explain how this theory is implemented. Therefore we need to rewrite the typing rules. It will also be possible to define type schemata.

- At last we look at the examples from this paper in the implemented syntax. Here we will see the reduction steps for recursion and corecursion. We will conclude this section with the example of partial streams, which is a coinductive type which depends on a value.

2. Coinductive Types

Inductive types are defined via their constructors. Coinductive types on the other hand are defined via their destructors. In the paper [APTS13] functions, which have coinductive types as their output, are implemented via copattern matching. In this paper streams are defined like the following:

```
record Stream A = { head : A,
                    tail : Stream A }
```

The A in the definition should be a concrete type. The type system in the paper don't has dependent types. What differentiate this from regular record types (for example in Haskell), is the recursive field tail. So they call it a recursive record. In a strict language without coinductive types we could never instantiate such a type, because to do this we already need something of type `Stream A` to fill in the field `tail`. To remedy this the paper defines copattern matching. With the help of copattern matching we can define functions which outputs expressions of type `Stream A`. As an example we look at the definition of repeat. This function takes in a value of type `Nat` and generates a stream which just infinitely repeats it.

```
repeat : Nat → Stream Nat
head (repeat x) = x
tail (repeat x) = repeat x
```

As you can see copattern matching works via observations i.e. we define what should be the output of the fields applied to the result of the function. These fields are also called observers, because we observe parts of the type. Because inhabitants of `Stream` are infinitely long we can't print out a stream. Because of this we also consider each expression with has a type, which is coinductive, as a value. To get a subpart of this value we have to use observers . For example we can look at the third value of `repeat 2` via `head (tail (tail (repeat 2)))` which should evaluate to 2. We can also implement a function which looks at the nth. value. Here it is:

```
nth : Nat → Stream A → A
nth 0     x = head x
nth (S n) x = nth n (tail x)
```

As you can see we use ordinary pattern matching on the left hand side and observers on the right hand side. `nth 3 (repeat 2)` will output `2` as expected. Functions can also be defined via a recursive record. It is defined like the following:

```
record A → B = { apply : A ⇝ B }
```

Here we differentiate between our defined function `A ⇾ B` and $\rightsquigarrow$ in the destructor. Constructor application or, as is the case here, destructor application is not the same as function application, like in Haskell. In the paper `f x` means `apply f x`. We will also use this convention in the following. In fact we already used it in the definitions of the functions `repeat` and `nth`. `nth 0 x = head x` is just a nested copattern. We can also write it with ‘apply‘ like so: `apply (apply nth 0) x = head x`. Here we use currying. So the first apply is the sole observer of type `Stream A ⇾ A` and the second of type `Nat ⇾ (Stream A ⇾ A)`.

3. Coinductive Types in dependent languages

In this section we will look how coinductive types are implemented in dependently typed languages. In dependently typed languages types can depend on values. The classical example for such a type is the vector. Vectors are like list, except their length is contained in their type. For example a vector of natural numbers of length 2 has type `Vec Nat 2`. This type depends on two things. Namely the type `Nat` and the value `2`, which is itself of type `Nat`. We can define vectors in coq like follows:

```
Inductive Vec (A : Set) : nat -> Set :=
  | Nil : Vec A 0
  | Cons : forall {k : nat}, A -> Vec A k -> Vec A (S k).
```

Contrary to a list the type constructor `Vec` has a second argument `nat`. This is the already mentioned length of the vector. A Vector has two constructors. One for the empty vector called `Nil` and one to append a element at the front of a vector called `Cons`. `Nil` just returns a vector of length `0`. And `Cons` gets an `A` and a vector of length `k`. It returns a vector of length `S k` (`S` is just the successor of k). This type can also be defined in agda like follows:

```
data Vec (A : Set) : ℕ → Set where
  Nil : Vec A 0
  Cons : {k : ℕ} → A → Vec A k → Vec A (suc k)
```

One advantage of vectors over list is that we can define a total function (a function which is defined for every input) which takes the head of a vector. This function can't be total for lists, because we can't know if the input list is empty. an empty list has no head. For vectors we can enforce this in coq like follow.

```
Definition hd {A : Set} {k : nat} (v : Vec A (S k)) : A :=
  match v with
  | Cons _ x _ => x
  end.
```

We just pattern match on `v`. The only patter is for the `Cons` constructor. The `Nil` constructor is a vector of length 0. But `v` has type `Vec A (S k)`. So it can't be a vector of length 0. In agda the function looks like follow.

```
hd : {A : Set} {k : ℕ} → Vec A (suc k) → A
hd (cons x _) = x
```

That terms can occur in types makes it necessary to ensure that function terminate. Otherwise type checking wouldn't be decidable. If we have a function `f : Nat → Nat` and we want to check a value `a` against a type `Vec (f 1)` we have to know what `f 1` evaluates to. So `f` has to terminate. We check termination in coq via a structural decreasing argument. An argument is structural decreasing, if it is structural smaller in a recursive call. Structural smaller means it is a recursive occurrence in a constructor. As an example we look at the definition of the natural numbers and the function for addition on them. We define the natural numbers in coq like follows:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

`O` is the constructor for 0 and `S` is the successor of its argument. Here the recursive argument to `S` is structural smaller than S applied to it i.e. `n` is structural smaller than `S n`. Then we can define addition like follows:

```
Fixpoint add (n m : nat) : nat :=
match n with
| O => m
| S p => S (add p m)
end.
```

In the recursive call the first argument is structural decreasing. `p` is smaller than `s p`. So coq accepts this definition. The classical example for a function where an argument is decreasing, but not structural decreasing is Quicksort. A naive implementation would be the following:

```
Fixpoint quicksort (l : list nat) : list nat :=
match l with
| nil => nil
| cons x xs => match split x xs with
              | (lower, upper) => app (quicksort lower) (cons x (quicksort upper))
              end
end.
```

Here `split` is just a function which gets a number and a list of numbers. It gives back a pair of two lists where the left list are all elements of the input list which are smaller than the input number and the right these which are bigger. It is clear that these lists can't be longer than the input list. So `lower` and `upper` can't be longer than `xs`. Here `xs` is structural smaller than the input `cons x xs`. So `lower` and `upper` are smaller than the input. Therefore we know that `quicksort` is terminating. But coq won't accept our code, because no argument is structural decreasing.

For coinductive types termination means that functions which produce them should be productive. If a function is productive it produces in each step a new part of the infinitely large coinductive type.

In section 3.1 we will look at the implementation in coq. There are two ways to define them. The older way uses positive coinductive types. This is known to violate subject reduction. Therefore it is highly discouraged to use them. To fix this the new way uses negative coinductive types . In section 3.2 we look at the implementation in agda. Agda also has the two ways of defining such types. One special thing about it, is that it implements copattern matching. To help agda with termination checking we can use sized types. We will explain them in section 3.2.3.

3.1. Coinductive Types in Coq

There are two approaches to define coinductive types in coq. The older one is described in 3.1.1. It works over constructors. Therefore they are called positive coinductive types. The newer and recommended one is described in section 3.1.2. They are defined over primitive records (a relatively new feature of coq). Therefore they are called negative coinductive Types.

3.1.1. Positive Coinductive Types

Positive coinductive types are defined over constructors in coq. The keyword `CoInductive` is used to indicate that we about to define a coinductive type. This is the only syntactical difference from the definition of inductive types. For example streams are defined like the following:

```
CoInductive Stream (A : Set) : Set :=
  Cons : A -> Stream A -> Stream A.
```

If this was an inductive type we couldn't generate a value of this type. To generate values of coinductive types coq uses guarded recursion. This checks if the recursive call to the function occurs as an argument to a coinductive constructor. In addition to the guard condition the constructor can only be nested in other constructors, fun or match expressions. With all of this in mind we can define `repeat` like the following:

```
CoFixpoint repeat (A : Set) (x : A) : Stream A := Cons A x (repeat A x).
```

Then we can produce the constant zero stream with `repeat nat 0`. If we used a normal coq function i.e. write `Fixpoint` instead of `CoFixpoint` coq wouldn't except our code. It rejects it, because there is no argument which is structural decreasing. `x` stays always the same. `CoFixpoint` on the other hand only checks the previously mentioned conditions. It sees the recursive call `repeat A x` occurs as an argument to constructor `Cons` of the coinductive type `Stream`. This constructor is also not nested. So our definition is accepted.

We can use the normal pattern matching of coq to destruct a coinductive type. We define `nth` like the following:

```
Fixpoint nth (A : Set) (n : nat) (s : Stream A) {struct n} : A :=
  match s with
    Cons _ a s' =>
    match n with 0 => a | S p => nth A p s' end
  end.
```

The guard condition is necessary to ensure every expression is terminating. If we didn't have the guard condition we could define the following:

```
CoFixpoint loop (A : Set) : Stream A = loop A.
```

Here the recursive call doesn't occur in a constructor. So the guard condition is violated. With this definition the expression `nth 0 loop` wouldn't terminate. `nth` would try to pattern match on `loop`. But to succeed in that `loop` has to unfold to something of the form `Cons a ?` which it never does. So `nth 0 loop` will never evaluate to a value. This would lead to undecidable type checking.

We illustrate the purpose of the other conditions on an example taken from [Chl13]. First we implement the function `tl` like so.

```
Definition tl A (s : Stream A) : Stream A :=
  match s with
  | Cons _ _ s' => s'
  end.
```

This is just one normal pattern match on `Stream`. If we didn't had the other condition we could define the following:

```
CoFixpoint bad : Stream nat := tl nat (Cons nat 0 bad).
```

This doesn't violate the guard condition. The recursive call `bad` is a argument to the constructor `Cons`. But the constructor is nested in a function. If we would allow this, `nth 0 bad` would loop forever. To understand why, we first unfold `tl` in `bad`. So we get

```
nth 0 (cofix bad : Stream nat :=
         match (Cons 0 bad) with
         | Cons _ s' => s'
         end)
```

We can now simplify this to just

```
nth 0 (cofix bad : Stream nat := bad)
```

After that `bad` isn't anymore an argument to a constructor. Here we can also see easily that the expression `cofix bad : Stream nat := bad` loops for ever. So we never get the value at position `0`.

An important property of typed languages is subject reduction. Subject reduction says if we evaluate an expression e_1 of type t to an expression e_2, e_2 should also be of type t. With positive coinductive types subject reduction is no longer valid. We illustrate this by Oury's counterexample [Our08]. First we define the codata type `U` as follows

```
CoInductive U : Set := In : U -> U.
```

We can now define a value of u with the following `Cofixpoint` like so

```
CoFixpoint u : U := In u.
```

This generates an infinite succession of `In`. We use the function `force` to force `U` to evaluate one step i.e. `x` becomes `In y`.

```
Definition force (x: U) : U :=
  match x with
```

```
    In y => In y
  end.
```

The same trick will be used to define `eq` which sates that `x` is definitional equal to `force x`.

```
Definition eq (x : U) : x = force x :=
  match x with
    In y => eq_refl
  end.
```

This first matches on x to force it, to reduce to `In y`. Then the new goal becomes `In y = force (In y)`. `force (In y)` evaluates to just `In y`, as it is just pattern matching on `In y`. So the final goal is `In y = In y` which can be shown by `eq_refl`. `eq_refl` is a constructor for =, where both sides of = are exactly the same. If we now instantiate `eq` with `u` we become `eq u`.

```
Definition eq_u : u = In u := eq u
```

But `u` is not definitional equal to `In u`. As mentioned above expression with a coinductive type are always values to prevent infinite evaluation. So `In u` is a value and `u` is also a value. But values are only definitional equal, if they are exactly the same. The next section will solve this problem through negative coinductive types.

3.1.2. Negative Coinductive Types

In coq 8.5. primitive records were introduced. With this it is now possible to define types over there destructors. So we can have negative , especially negative coinductive, types in coq. With primitive records we can define streams like the following:

```
CoInductive Stream (A : Set) : Set :=
  Seq { hd : A; tl : Stream A }.
```

Now we can define `repeat` over the fields of `Stream`.

```
CoFixpoint repeat (A : Set) (x : A) : Stream A :=
  {| hd := x; tl := repeat A x|}.
```

To define `repeat` we must define what is the head of the constructed stream and what it is tail. The guard condition says now that corecursive occurrences must be guarded by a record field. We can see that the corecursive call `repeat` is a direct argument to the field `tl` of the corecursive type `Stream A`. This means coq accepts the above definition. If we want to access parts of a stream we use the destructors `hd` and `tl`. With them we can define `nth` again for the negative stream.

```
Fixpoint nth (A : Set) (n : nat) (s : Stream A) : list A :=
  match n with
  | 0 => s.(hd A)
  | S n' => nth A n' s.(tl A)
  end.
```

With negative coinductive types we can't form the above mentioned counterexample to subject reduction anymore, because we can't pattern match on negative types.

Oury's example becomes.

```
CoInductive U := { out : U }.
```

`U` is now defined over its destructor `out`, instead of its constructor `in`. Then `in` becomes just a function. In Fact its just a definition, because we don't recurse or corecurse on it.

```
Definition In (y : U) : U := {| out := y |}.
```

We define it over the only field `out`. When we put a `y` in then we get the same `y` out. We can also again define `u`.

```
CoFixpoint u : U := {| out := u |}.
```

With coinductive types it is know possible to define the pi type (the depend function type).

```
CoInductive Pi (A : Set) (B : A -> Set) := { Apply (x : A) : B x }.
```

The pi type is defined over its destructor `Apply`. If we evaluate `Apply` on a value of `Pi` (which is a function) and an argument, we get the result i.e. we apply the value to the function. It looks like the pi type becomes definable in coq. But we are cheating. The type of `Apply` is already a pi type. This is because we identify constructors and destructors with functions. We will see that the theory of the paper avoids this identification. To define a function we use `CoFixpoint`. As a simple non recursive, non dependent example we use the function `plus2`.

```
CoFixpoint plus2 : Pi nat (fun _ => nat) :=
  {| Apply x  := S (S x) |}.
```

If we apply (i.e. call the destructor `Apply`) a `x` to plus2 it gives back `S (S x)`. Which is twice the successor on `x`. So we add 2 to `x`. We use _ here because `plus2` is not a dependent function i.e. the result type `nat` doesn't depend on the input value. To define functions with more than one argument we just use currying i.e. we use the type `Pi` as the second argument to `Pi`. For example a 2-ary non-dependent function from `A` and `B` to `C` would have type `Pi A (fun _ => Pi B (fun _ => C))`. It would be fortunate if we could define `plus` like the following:

```
CoFixpoint plus : Pi nat (fun _ => Pi nat (fun _ => nat)) :=
  {| Apply := fun (n : nat)  =>
       match n with
       | O => {| Apply (m : nat) := m |}
       | S n' => {| Apply m := S (Apply _ _ (Apply _ _  plus n') m) |}
       end
  |}.
```

But coq doesn't accept this definition. The guard condition is violated. `plus n'` is not a direct argument of the field `Apply`. The definition should terminate because we are decreasing `n` and the case for `0` is accepted. In the case for `0`, there is no recursive call.

We can also define a dependent function. We define append2Units like follows

```
CoFixpoint append2Units : Pi nat
                             (fun n => Pi (Vec unit n)
                                         (fun _ => Vec unit (S (S n)))) :=
  {| Apply n := {| Apply v := Cons _ tt (Cons _ tt v) |} |}.
```

This just appends 2 units at a vector of length `n`. Here the second argument and the result depend on the first argument i.e. the first argument is the length of the input vector and the output vector is this length plus two.

3.2. Coinductive Types in Agda

In agda coinductive types where first also introduced as positive types. In the section 3.2.1 we will look at them in detail. In section 3.2.2 we describe the correct way to implement coinductive types in agda. There are functions which terminate but are rejected by the type checker. In fact in any total language there have to be such functions. We can show that by trying to list all total functions. The following table lists functions per row. The columns say what the output of the functions for the given input is.

	1	2	3	4	…
f_1	2	7	8	6	…
f_2	4	4	6	19	…
f_3	6	257	1	2	…
f_4	7	121	23188	2313	…
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We can now define a function $g(n) = f_n(n) + 1$ this function is total and not in the list, because it is different to any function in the list for at least one input. To allow more functions we can use an unique feature of agda, sized types. They are described in section 3.2.3.

3.2.1. Positive Coinductive Types in Agda

Agda doesn't has a special keyword to define coinductive types like coq. It uses the symbol ∞ to mark arguments to constructors as coinductive. This symbol says that the computation of arguments of this type are suspended. ∞ is just a type constructor. So agda ensures productivity over type checking. We define streams like so.

```
data Stream (A : Set) : Set where
  cons : A → ∞ (Stream A) → Stream A
```

Here the second argument to `cons` is marked with ∞. This is the tail of the stream. Because it is infinitely long (we don't have a constructor of an empty stream) we can't compute it completely, so we suspend the computation. We can delay a computation with the constructor ♯ and force it with the function ♭. Their types are given below.

```
♯_ : ∀ {a} {A : Set a} → A → ∞ A
♭  : ∀ {a} {A : Set a} → ∞ A → A
```

We can now again define our usual functions. We begin with `repeat`.

```
repeat : {A : Set} → A → Stream A
repeat x = cons x (♯ (repeat x))
```

We first apply `cons` to `x`. So the head of the stream is `x`. We then apply it to the corecursive call `repeat`. So the tail will be a repetition of `xs`. We have to call the `repeat` with ♯ to suspend the computation. Otherwise the code doesn't type check. If we would write this function without ♯ on a stream which has no ∞ on the second argument of `cons`, the function would run forever. In fact the termination checker won't allow us to write such a function. We can also write `nth` again, which consumes a stream.

```
nth : {A : Set} → ℕ → Stream A → A
nth 0       (cons x _)  = x
nth (suc n) (cons _ xs) = nth n (♭ xs)
```

Here we have to use ♭ on the right hand side of the second case, to force the computation of the tail of the input stream. We have to do that because `nth` wants a stream. It doesn't want a suspended stream. Productivity on coinductive types like stream is checked by only allowing non decreasing recursive calls behind the ♯ constructor.

3.2.2. Negative Coinductive Types in Agda

In agda we can also define negative coinductive types. This is the recommended way. Agda implements the previously mentioned copattern matching. We can define a record with the keyword `record`. We use the keyword `coinductive` to make it possible to define recursive fields. Stream is defined like the following:

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : Stream A
```

A Stream has 2 fields. `hd` is the head of the stream. It has type `A`. `tl` is the tail of the stream. It is another stream, so it has type `Stream A`. `tl` is a recursive field. So agda wouldn't accept the definition without `coinductive`. Stream can never be empty. Every stream has a head (a field `hd`) and an empty stream wouldn't have a head. So the tail of a stream can never be empty. Therefor every stream is infinitely long. We can now define `repeat` with copattern matching.

```
repeat : ∀ {A : Set} → A → Stream A
hd (repeat x) = x
tl (repeat x) = repeat x
```

We have to copattern match on every field of `Stream`, namely `hd` and `tl`. Because agda is total it won't accept non-exhaustive (co)pattern matches like Haskell. First we define what the head of `repeat x` is. We just repeat `x` infinitely often. So every element of the steam is `x`, including the head. Therefor we just write `x`. In the second and last copattern we define what the tail of the stream is. The tail is just `repeat x`. Infinitely often repeated `x` is the same as x and then infinitely repeated `x`. We can use normal pattern matching and the destructors for functions which consume streams. We define `nth` like the following:

```
nth : ∀ {A : Set} → ℕ → Stream A → A
nth zero s = hd s
nth (suc n) s = nth n (tl s)
```

Here we just pattern match on the first argument (excluding the implicit argument of the type). If it is zero the result is just the head of the stream. If it is $n+1$ the result is the recursive call of `nth` on `n` and `tl s`. Agda accepts this code, because it is structural decreasing on the first (or second if we count the implicit) argument.

We can also define the pi type. We use _$_ as the apply operator. This operator is taken from Haskell.

```
record Pi (A : Set) (B : A → Set) : Set where
  field _$_ : (x : A) → B x
  infixl 20 _$_
open Pi
```

Like in coq we are using the first-class pi type to define the pi type. We can also define a function which adds 2 to a number `plus2` in agda.

```
plus2 : ℕ →' ℕ
plus2 $ x = suc (suc x)
```

We just use copattern matching to define it. If we apply a `x` to `plus2` we get `suc (suc x)`. →′ is just the non-dependent function it is defined using our pi type. Here it is:

```
_→'_ : Set → Set → Set
A →' B = Pi A (λ _ → B)
infixr 20 _→'_
```

In agda it becomes possible to define plus. We just use nested copattern matching.

```
plus : ℕ →' ℕ →' ℕ
plus $ 0       $ m = m
plus $ (suc n) $ m = suc (plus $ n $ m)
```

If we change →′ to → and remove $ we get the standard definition for plus in agda. We can also define a dependent function `repeatUnit` like follow

```
repeatUnit : Pi ℕ (λ n → Vec ⊤ n)
repeatUnit $ 0     = nil
repeatUnit $ suc n = tt :: (repeatUnit $ n)
```

This function gives back a vector with the length of the input, where every element is unit.

3.2.3. Termination Checking with Sized Types

They are many functions, which are total but are not accepted by Agda's termination checker. For example we could try to define division with rest on natural numbers like the following:

```
_/_ :  ℕ → ℕ → ℕ
zero / y = zero
suc x / y = suc ( (x - y) / y)
```

The problem with this definition is that agda doesn't know that $\mathtt{x} - \mathtt{y}$ is smaller than $\mathtt{x} + 1$, which is clearly the case (`x` and `y` are positive). This definition would work perfectly fine in a language without termination checking (like Haskell). Agda only checks if an argument is structurally decreasing. Here it is neither the case for `x` nor for `y`.

To remedy this problem sized types where introduced first to mini-agda (a language specifically developed to explore them) by [Abe10]. Later they got introduced to agda itself. Sized types allow us to annotate data with their size. Functions can use this sizes to check termination and productivity.

We can now define the natural numbers depending on a size argument.

```
data ℕ (i : Size) : Set where
  zero : ℕ i
  suc : ∀{j : Size< i} → ℕ j → ℕ i
```

The natural number now depends on a size `i`. The constructor `zero` is of arbitrary size `i`. `suc` gets a size `j` which is smaller than `i`, a natural number of size `j` and gives back a natural number of size `i`. This means the size of the input is smaller than the size of the output. For inductive types, a size is an upper bound on the number of constructors. With `suc` we add a constructor so the size has to increase `i`. We can now define subtraction on these sized natural numbers.

```
_-_ : {i : Size} → ℕ i → ℕ ∞ → ℕ i
zero    - _       = zero
n       - zero    = n
(suc n) - (suc m) = n - m
```

Through the sized annotations, we know now that the result isn't larger than the first input. ∞ means that the size isn't bound. If the first argument is zero the result is also zero, which has the same type. If the second argument is zero we return just the first. In the last case both arguments are non-zero. We call subtraction recursively on the predecessors of the inputs. Here the size and both arguments are smaller. So the function terminates. Tough the type is smaller then i, the result type checks because sizes are upper bounds. We can now define division.

```
_/_ : {i : Size} → ℕ i → ℕ ∞ → ℕ i
zero  / _ = zero
suc x / y = suc ( (x - y) / y)
```

From the definition of `suc` we know that the size of `x` is smaller than `i`. Because the result of `-` has the same size as it's first input (here `x`), we also know that (`x - y`) has the same size as `x`. Therefor (`x - y`) is smaller than `suc x` and the function is decreasing on the first argument. Also, agda accepts this definition.

We can also use sized types for coinductive types. To show this we will define the hamming function. This produces a stream of all composites of two and three in order. First we will define the sized stream type.

```
record Stream (i : Size) (A : Set) : Set where
  coinductive
  field
    hd : A
    tl : ∀ {j : Size< i} → Stream j A
open Stream
```

This stream has a new parameter of type `Size`. This size gives the minimal definition depth of the stream. The definition depth says how often we can destruct the stream without diverging. If we take the tail of an stream, the output streams depth would be one smaller. Because in agda coinductive types can't have indexes, we can only say that its depth is smaller. We will now define some helper functions for the hamming function. First we need a cons function.

```
cons : {i : Size} {A : Set} → A -> Stream i A → Stream i A
hd (cons x _)  = x
tl (cons _ xs) = xs
```

This just appends an element at the front of the stream. Because the output streams depth is larger than the input and the size is a minimum, we can give the output the same size parameter as the input. Now we will define map over stream.

```
map : {A B : Set} {i : Size} → (A → B) → Stream i A → Stream i B
hd (map f xs) = f (hd xs)
tl (map f xs) = map f (tl xs)
```

This function just changes the contend of the stream so the size stays the same. The last helper function we need is the merge function.

```
merge : {i : Size} → Stream i ℕ → Stream i ℕ → Stream i ℕ
hd (merge xs ys) = hd xs ⊓ hd ys
tl (merge xs ys) = if ⌊ hd xs ≤? hd ys ⌋
                   then cons (hd ys) (merge (tl xs) (tl ys))
                   else cons (hd xs) (merge (tl xs) (tl ys))
```

This functions just merges two streams. It always compares one element of each stream with each other and puts the bigger after the smaller. This is clear in the case for `hd` (⊔ is just the binary minimum function in agda). in the `tl` case we just compare the heads of the stream and construct the tail with `cons` accordingly. Both input streams have a minimal definition depth of `i`. Because `cons` isn't destructing the stream (the minimal depth doesn't get smaller) we can say that the minimal depth of the output also won't get smaller. With all this function we can now define the ham function. Here it is:

```
ham : {i : Size} → Stream i ℕ
hd ham = 1
tl ham = (merge (map (_*_ 2) ham) (map (_*_ 3) ham))
```

None of the used function is destructing the stream, so this definition gets accepted.

4. Type Theory based on dependent Inductive and Coinductive Types

In the paper [BG16] a type theory, where inductive types and coinductive types can depend on values, is developed. For example we can, in contrast to the coinductive types of coq and agda, define streams which depend on their definition length. The theory differentiates types from terms. We don't have infinite universes, where a term in universe n has a type in universe $n+1$(This is how it is done in coq [ST14] and agda [agd]). Therefore types can only depend on values, not on other types. We only have functions on the type level. These functions abstract over terms. For example $\lambda x.A$ is a type where all occurrences of the term variable x in A are bound. We will see that functions are definable on the term level. We can apply types to terms. For example $A@t$ means we apply the term A to x. Every type has a kind. A kind is either $*$ or $\Gamma \multimap *$. Here Γ is a context, which states to what terms we can apply the type. For example we can apply A of kind $(x:B) \multimap *$ only to a term of type B. If we apply it to t of type B, we get a type of kind $*$. We write $\multimap$ instead of $\rightarrow$ to indicate, that these are not functions. We can also apply a term to another term. For example $t@s$ means we apply the term t to the term s. Terms also can depend on contexts. For example if we have a term t of type $(x:A) \multimap B$ and apply it to a term s of type A we get a term of type B. We can also define our own types. $\mu(X:\Gamma \multimap *;\vec{\sigma};\vec{A})$ is an inductive type and $\nu(X:\Gamma \multimap *;\vec{\sigma};\vec{A})$ is a coinductive type. X is a variable which stands for the recursive occurrence of the type. It has the same kind $\Gamma \multimap *$ as the defined type. The $\vec{A}$ can contain this variable. There are also contexts $\vec{\Gamma}$, which are implicit in the paper. σ_k and A_k can contain variables from Γ_k. σ_k is a context morphism from Γ_k to Γ. A context morphism is a sequence of terms, which depend on Γ_k and instantiate Γ. $\vec{\sigma}$, $\vec{A}$ and $\vec{\Gamma}$ are of the same length.

In this theory we can define partial streams on some type A like the following:

$$
\begin{aligned}
&\text{PStr } A := \nu(X:(n:\text{Conat}) \multimap *;(\text{succ}@n,\text{succ}@n);(A,X@n))\\
&\text{with } \Gamma_1 = (n:\text{Conat}) \text{ and } \Gamma_2 = (n:\text{Conat})
\end{aligned}
$$

Here `succ` is the successor on co-natural numbers. Co-natural numbers are natural numbers with one additional element, infinity. See 6.2 for their definition. Here the first destructor is the head. It becomes a stream with length succ@N and returns an A. The second destructor is the tail. It becomes also a stream of length succ@N. It gives back an $X@n$, which is a stream of length n. We can also define the Pi type from A to B, where B can depend on A.

$$\begin{aligned}&\Pi x : A.B := \nu(\_ : *;\epsilon_1;B)\\ &\text{with } \Gamma_1 = (x : A)\end{aligned}$$

By _ we mean, we are ignoring this variable. ϵ_1 is one empty context morphism. So the only destructor gives back a B which can depend on x of type A. It is the function application.

To construct an inductive types we use constructors (written $\alpha_k^{\mu(X:\Gamma \rightarrow\!\triangleright *;\vec{\sigma};\vec{A})}$ in the paper, which is the k-th constructor of the given type). We can destruct it with recursion (written $\mathtt{rec}\ \overrightarrow{(\Gamma_k.y_k).g_k}$). Coinductive type work the other way around. We destruct them with destructors (written $\xi_k^{\nu(X:\Gamma \rightarrow\!\triangleright *;\vec{\sigma};\vec{A})}$) and construct them with corecursion (written $\mathtt{corec}\ \overrightarrow{(\Gamma_k.y_k).g_k}$).

We will give the rules for the theory in section 5.3 and a detailed explanation of the reduction in 5.4.

5. Implementation

In this section we look at the implementation details. We use the functional programming language Haskell for implementing the theory. Haskell is a pure language. This means functions which aren't in the IO monad have no side effects. The only IO we are doing is reading a file and as the last step printing it. Because everything between is pure, we can test it without bordering on side effects. Another feature of Haskell, which will be get useful in our implementation is pattern matching. We will see its usefulness in section 5.3.

In section 5.1 we will develop the abstract syntax of our language from the raw syntax in the paper. Then we rewrite the typing rules in 5.3. At last we look at the implementation of the reduction in 5.4

5.1. Abstract Syntax

In the following we will scratch out the abstract syntax. In contrast to [BG16] we can't write anonymous inductive and coinductive types. We will give every inductive and coinductive type a name. They will be defined via declarations. In these declarations we will give, their constructors/destructors. They will also be given names. In [BG16] they are anonymous. We can then refer to the previously defined types. We will described declarations in section 5.1.1. We will also be able to bind expressions to names. In section 5.1.2 we will define the syntax of expressions. This will mostly be in one to one correspondence with the syntax of [BG16]. Note however that we use the names of the constructors instead of anonymous constructors together with their type and number. Also the order of the matches in `rec` and `corec` is irrelevant. We use the names of the Con/Destructors to identify them. In the following section 6 we will see how the examples from the paper look in our concrete syntax.

5.1.1. Declarations

The abstract syntax is given in figure 5.1. With the keywords data and codata we define inductive and coinductive types respectively. After that we will write the name. We can only use names which aren't used already. Behind that we can give a parameter context. This is a type context. These types are not polymorphic. They are merely macros to make the code more readable and allow the definition of

N	$:=$	$[A-Z][a-zA-Z0-9]*$	Names for types, constructors and destructors
n	$:=$	$[a-z][a-zA-Z0-9]*$	Names for expressions
EV	$:=$	$x,y,z,\ldots$	Expression variables
TV	$:=$	$X,Y,Z,\ldots$	Type expression variables
PV	$:=$	$A,B,C,\ldots$	Parameter variables
EC	$:=$	$\Diamond$	Expression Context
	$\mid$	$(EV : TV(,EV : TV)*)$	
PC	$:=$	$\langle\rangle$	Parameter Context
	$\mid$	$\langle(PV : EC \to \text{ Set})*\rangle$	
$Decl$	$:=$	data $N\, PC : (EC \to)?$ Set where $(N : (EC \to)?TypeExpr \to N\, Expr*)*$	Declarations
	$\mid$	codata $N\, PC : (EC \to)?$ Set where $(N : (EC \to)?N\, Expr* \to TypeExpr)*$	
	$\mid$	$n\, PC\, EC = Expr$	

Figure 5.1.: Syntax for declarations

nested types. If we want to use these types we have to fully instantiate this context. These types can occur everywhere in the definition where a type is expected. A (co)inductive type can have a context, which is written before an arrow. `Set` stands for type (or * in the paper). If a type don't has a context we omit the arrow. We will also give names to every constructor and destructor. These names have to be unique. Constructors and destructors also have contexts. Additionally they have one argument which can has a recursive occurrence of the type we are defining. A constructor gives back a value of the type, where its context is instantiated. This instantiation corresponds to the sigmas in the paper. If we write a name before an equal sign we can bind the following expression to the name. Every such defined name can depend on a parameter context and an argument context. We write the parameter context like in the case for data types behind the name. After that we can give a term context between round parenthesis.

The declarations in Figure 5.1 correspond to $\rho(X : \Gamma \multimap *; \vec{\sigma}; \vec{A}) : \Gamma \multimap *$ as follows:

- The first N is X
- The other N will be used later for $\alpha_1^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}, \alpha_2^{\mu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}, \ldots$ in the case of inductive types and $\xi_1^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}, \xi_2^{\nu(X:\Gamma\multimap *;\vec{\sigma};\vec{A})}, \ldots$ in the coinductive case
- The *TypExpr* are the $\vec{A}$

- The *Expr** are the $\vec{\sigma}$
- The first *EC* is Γ
- The other *EC* stand for $\Gamma_1,\ldots,\Gamma_m$

To parse the abstract syntax we use Megaparsec. The parser generates an abstract syntax tree, which is given for declarations in Listing 1. The field `ty` in `ExprDef` is used later in type checking. The parser just fills them in with `Nothing`. data and codata definitions are both saved in `TypeDef`. The Haskell type `OpenDuctive` contains all the information for inductive and coinductive types. It corresponds to μ and ν in the paper. We use an `OpenDuctive` where the field `inOrCoin` is `IsIn` for μ and an `OpenDuctive` where the field `inOrCoin` is `IsCoin` for ν. The Haskell type `StrDef` ensures that the sigmas, as and gamma1s have the same length. We omit the implementation details for the parser, because we are manly focused on type checking.

```
data Decl = ExprDef { name :: Text
                    , tyParameterCtx :: TyCtx
                    , exprParameterCtx :: Ctx
                    , expr :: Expr
                    , ty :: Maybe Type
                    }
          | TypeDef OpenDuctive
          | Expression Expr

data OpenDuctive = OpenDuctive { nameDuc :: Text
                               , inOrCoin :: InOrCoin
                               , parameterCtx :: TyCtx
                               , gamma :: Ctx
                               , strDefs :: [StrDef]
                               }

data StrDef = StrDef { sigma :: [Expr]
                     , a :: TypeExpr
                     , gamma1 :: Ctx
                     , strName :: Text
                     }
```

Listing 1: Implementation of the abstract syntax of fig. 5.1

5.1.2. Expressions

The abstract syntax for expression is given in figure 5.2. We will separate expression in expressions for terms and expressions for types. There are given as regular expressions in `Expr` and `TypeExpr` respectively.

An `Expr` is either a `rec`, a `corec`, a con/destructor, an application @,the only primitive unit expression $\Diamond$ or a variable. With the keyword `rec` we can destruct an inductive type. We write `N ParInst? to TypeExrp`, where `N` is a previously defined inductive type and `ParInst?` the instantiation of its parameter context, after `rec` to facilitate type checking. It says we want to destruct an inductive type to some other

ParInst	:=	$\langle$*TypeExpr*(,*TypeExpr*)*$\rangle$	Instantiations for paramter contexts
ExprInst	:=	(*Expr*(,*Expr*)*)	Instantiations for expression contexts
Expr	:=	rec *N ParInst*? to *TypeExpr* where *Match**	expression
	|	corec *TypeExpr* to *N ParInst*? where *Match**	
	|	*Expr* @ *Expr*	
	|	$\lozenge$	
	|	*EV*	
	|	*n ParInst ExprInst*	
Match	:=	*N EV** = *Expr*	match
TypeExpr	:=	(*EV* : *TypeExpr*).*TypeExpr*	Type expressions
	|	*TypeExpr* @ *Expr*	
	|	Unit	
	|	*TV*	
	|	*N ParInst*?	

Figure 5.2.: Syntax for expressions

type . We have to list all the constructors above one another. For each constructor we write an expression behind the equal sign, which should be of type `TypeExpr` which we have given above. In this expression we can use variables given in the match expression. The last one is the recursive occurrence. With the keyword `corec` we can do the same thing to construct a coinductive type. Here we have to swap the `N ParInst?` and the `TypeExpr` and list the destructors. All con/destructors have to be instantiate with all variables in the parameter contexts of their types. This is done by giving types of the expected kinds separated by ',' enclosed in ⟨ and ⟩. The variables are separated in local variables and global variables. Global variables refer to previously defined expressions. We have to fully instantiate they parameter contexts and their expression contexts. We can also apply an expression to another with @. This application is left associative. So if we write `t @ s @ v` we mean `(t @ s) @ v`.

The `typeExpr` is either the unit type `Unit`, a lambda abstraction on types, an application or a variable. In the lambda expression we have to give the type of the variable. We apply a type to a term (types can only depend on terms) with @. As in the case of term application this is also left associative. The unit type is the only primitive type expression.

The generated abstract syntax tree is given in listing 2. The variables for expressions are separated in `LocalExprVar` and `GlobalExprVar`. `LocalExprVar` should refer to variables which are only locally defined i.e. in `Rec` and `Corec`. We use de Bruijn indexes for them. This facilitates substitution which we will describe in section 5.2. `GlobalExprVar` refers to variables from definitions. Here we just use names. We do the same thing for `LocalTypeVar` and `GlobalTypeVar`. In the abstract syntax tree we use anonymous constructors like in the paper. We combine them to the Haskell constructor `Structor`. We know from the field `ductive` if it is a constructor or a destructor. The types in field `parameters` are to fill in the parameter context of the field `ductive`. The field `nameStr` in `Constructor` and `Destructor` are just for printing. We combine `rec` and `corec` to `Iter`.

5.2. Substitution

In the following we will write $t[s/x]$ for "substitute every free occurrences of x in t by s". Substitution is done in the module `Subst.hs`. We use de Bruijn indexes [DB72] for bound variables to facilitate substitution. With this method every bound variable is a number instead of a string. The number says where the variable is bound. To find the binder of a variable we go outwards from it and count every binder until we reach the number of the variable. For example $\lambda.\lambda.\lambda.1$ says that the variable is bound by the second binder (we start counting at zero). This would be the same as $\lambda x.\lambda y.\lambda z.y$. This means we never have to generate fresh names. We just shift the free variables in the term with which we substitute by one, every time we encounter a binder. This shifting is done in the module `ShiftFreeVars.hs`. We

```
data TypeExpr = UnitType
              | TypeExpr :@ Expr
              | LocalTypeVar Int Bool Text
              | Parameter Int Bool Text
              | GlobalTypeVar Text [TypeExpr]
              | Abstr Text TypeExpr TypeExpr
              | Ductive { openDuctive :: OpenDuctive
                        , parametersTyExpr :: [TypeExpr]}

data Expr = UnitExpr
          | LocalExprVar Int Bool Text
          | GlobalExprVar Text [TypeExpr] [Expr]
          | Expr :@: Expr
          | Structor { ductive :: OpenDuctive
                     , parameters :: [TypeExpr]
                     , num :: Int
                     }
          | Iter { ductive :: OpenDuctive
                 , parameters :: [TypeExpr]
                 , motive :: TypeExpr
                 , matches :: [([Text],Expr)]
                 }
```

Listing 2: Implementation of the abstract syntax of fig. 5.2

also want to be able to substitute multiple variables simultaneously. If we would just substitute one term after another we could substitute into a previous term. For example the substitution $x[y/x][z/y]$ would yield z if we substitute sequential and y if we substitute simultaneously. To make simultaneous substitution possible every local variable has a boolean flag. If this flag is set to true substitution won't substitute for that variable. So for simultaneous substitution we just set this flag to true for all terms with which we want to substitute. Then we substitute with them. In the last step we just have to set the flags to false in the result. This setting(marking of the variables) is done in the module `Mark.hs`.

5.3. Typing rules

A typing rule says that some expression or declaration is of some type, given some premises. If we can for every declaration or expression form a tree of such rules with no open premises, our program type checks. We have to rewrite the typing rules of the paper, to get rules which are syntax directed. Syntax directed means we can infer from the syntax alone what we have to check next i. e. which rule with which premises we have to apply. In the paper their are rules containing variables in the premises where their type isn't in the conclusion. So if we want to type-check something which is the conclusion of such a rule we have no way of knowing what these variables are.

We don't need the weakening rules because we can lookup a variable in a context. So we ignore them in our implementation.

The order in **TyCtx** isn't relevant so we can use a map for it. In the code we use a list, because the names of the variables are the index of their type in the context. The order of **Ctx** is relevant because types of later variables can refer to former variables and application instantiate the first variable in **Ctx**. We add a new context for data types. We also need a context for the parameters. **Ctx** can contain variables from this context, but not from **TyCtx**.

We also rewrite the rules which are already syntax-directed to rules which work on our syntax. We will mark semantic differences in the rewritten rules gray. We use variables Φ,Φ',Φ_1,Φ_2,... for parameter contexts, Θ,Θ',Θ_1,Θ_2,... for type variable contexts and Γ,Γ',Γ_1,Γ_2,... for term variable contexts. The judgments in our rules are of one of the following form.

- $\Phi \mid \Theta \mid \Gamma \vdash \Theta'$ - The type variable context Θ' is well formed in the combined context $\Phi \mid \Theta \mid \Gamma$.
- $\Phi \mid \Theta \mid \Gamma \vdash \Gamma'$ - The term variable context Γ' is well formed in the combined context $\Phi \mid \Theta \mid \Gamma$.
- $\Phi \mid \Theta \mid \Gamma \vdash \Phi'$ - The parameter variable context Φ' is well formed in the combined context $\Phi \mid \Theta \mid \Gamma$.
- $A \longrightarrow_T^* B$ - The type A fully evaluates to type B.
- $A \equiv_\beta B$ - The type A is computational equivalent to type B.
- $\Phi \mid \Theta \mid \Gamma \vdash A : \Gamma_2 \rightarrowtriangle *$ - The type A is well formed in the combined context $\Phi \mid \Theta \mid \Gamma$ and can be instantiated with arguments according to context Γ_2.
- $\Phi \mid \Theta \mid \Gamma \vdash t : \Gamma_2 \rightarrowtriangle A$ - The term t is well formed in the combined context $\Phi \mid \Theta \mid \Gamma$ and can be instantiated with arguments according to context Γ_2. After this instantiation it is of type A, where the arguments are substituted in A.
- $\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2$ - The context morphism σ is a well-formed substitution for Γ_2 with terms in context Γ_1 in parameter context Φ.

We will write $\vdash$ for $\Phi \mid \Theta \mid \Gamma \vdash$ where Φ,Θ and Γ are arbitrary and aren't referred to by the right hand side.

In the module `TypeChecker` we will implement the following rules. It defines a monad `TI` which can throw errors and has a reader on the contexts in which we are type checking. To add something to a context we use the function `local`. This function gets a function to change the current content of the reader monad and executes a reader on this changed context in the current monad.

5.3.1. Context rules

The rules for valid contexts are already syntax directed so we take just them.

$$\frac{}{\vdash \emptyset \quad \mathbf{TyCtx}} \qquad \frac{\vdash \Theta \quad \mathbf{TyCtx} \qquad \vdash \Gamma \quad \mathbf{Ctx}}{\vdash \Theta, X : \Gamma \rightarrowtriangle * \quad \mathbf{TyCtx}}$$

$$\frac{}{\vdash \emptyset \quad \mathbf{Ctx}} \qquad \frac{\mid \emptyset \mid \Gamma \vdash A : *}{\vdash \Gamma, x : A \quad \mathbf{Ctx}}$$

In the rules for valid contexts we ensure that the types in the context can not depend on **TyCtx**. Note however that they can depend on **ParCtx**. This ensures that only strictly positive types are possible.

We also need new rules for checking if a parameter context is valid.

$$\frac{}{\vdash \emptyset \quad \mathbf{ParCtx}} \qquad \frac{\vdash \Phi \quad \mathbf{ParCtx} \qquad \vdash \Gamma \quad \mathbf{Ctx}}{\vdash \Phi, X : \Gamma \rightarrowtriangle * \quad \mathbf{ParCtx}}$$

This are structural the same rule as this for **TyCtx**. The difference is that **ParCtx** and **TyCtx** are used differently in the other rules, as we have already seen in the rule for **Ctx**.

We use the notation $\Theta(X) \rightsquigarrow \Gamma \rightarrowtriangle *$ for looking up the type variable X in type context Θ yields type $\Gamma \rightarrowtriangle *$. We add 2 rules for looking up something in a type context. They are:

$$\frac{\vdash \Theta \quad \mathbf{TyCtx} \qquad \vdash \Gamma \quad \mathbf{Ctx}}{\Theta, X : \Gamma \rightarrowtriangle *(X) \rightsquigarrow \Gamma \rightarrowtriangle *} \qquad \frac{\vdash \Gamma_1 \quad \mathbf{Ctx} \qquad \Theta(X) \rightsquigarrow \Gamma_2 \rightarrowtriangle *}{\Theta, Y : \Gamma_1 \rightarrowtriangle *(X) \rightsquigarrow \Gamma_2 \rightarrowtriangle *}$$

Here Y and X are different variables.

The rules for looking up something in a parameter context are principally the same.

$$\frac{\vdash \Phi \quad \mathbf{ParCtx} \qquad \vdash \Gamma \quad \mathbf{Ctx}}{\Phi, X : \Gamma \rightarrowtriangle *(X) \rightsquigarrow \Gamma \rightarrowtriangle *} \qquad \frac{\vdash \Gamma_1 \quad \mathbf{Ctx} \qquad \Phi(X) \rightsquigarrow \Gamma_2 \rightarrowtriangle *}{\Phi, Y : \Gamma_1 \rightarrowtriangle *(X) \rightsquigarrow \Gamma_2 \rightarrowtriangle *}$$

Respectively the notation $\Gamma(x) \rightsquigarrow A$ means looking up the term variable x in term context Γ yields type A. The rules for term contexts are:

$$\frac{\vdash \Gamma \quad \mathbf{Ctx} \qquad \Gamma \vdash A : *}{\Gamma, x : A(x) \rightsquigarrow A} \qquad \frac{\Gamma(x) \rightsquigarrow A \qquad \Gamma \vdash B : *}{\Gamma, y : B(x) \rightsquigarrow A}$$

5.3.2. Beta-equivalence

Two types are beta equivalent if they evaluate to the same type. Because our language is deterministic this just means if we fully evaluate both of them they are alpha equivalent. Alpha equivalence means we can substitute some variables in both of them and get the same type. So we first need to define rules which say what full evaluation means. We write $A \longrightarrow_T^* B$ for evaluating A as long as it is possible yields B.

The rules are:

$$\frac{\neg \exists B : A \longrightarrow_T B}{A \longrightarrow_T^* A} \qquad \frac{A \longrightarrow_T B \qquad B \longrightarrow_T^* C}{A \longrightarrow_T^* C}$$

$\longrightarrow_T$ is defined in section 5.4.

We can then introduce a new rule for beta-equivalence.

$$\frac{A \longrightarrow_T^* A' \qquad B \longrightarrow_T^* B' \qquad A' \equiv_\alpha B'}{A \equiv_\beta B}$$

This rule says if A evaluates to A', B to B' and A' and B' are alpha equivalent, then A and B are beta equivalent. In the implementation $\equiv_\alpha$ is trivial, because we use *de Bruijn indices*.

We also add some rules to check if two contexts are the same.

$$\frac{}{\emptyset \equiv_\beta \emptyset} \qquad \frac{\Gamma_1 \equiv_\beta \Gamma_2 \qquad A \equiv_\beta B}{\Gamma_1, x : A \equiv_\beta \Gamma_2, y : B}$$

5.3.3. Unit type and expression introduction

The paper defines one rule for the unit type and one for the unit value. These are.

$$\frac{}{\vdash \top : *}\ (\top\textbf{-I}) \qquad \frac{}{\vdash \Diamond : \top}\ (\top\textbf{-I})$$

The first rule says that the type $\top$ has always an empty context. The second rule says its value $\Diamond$ is always of type $\top$. These rules get rewritten to.

$$\frac{}{\Phi \mid \Theta \mid \Gamma \vdash \text{Unit}{:}*}\ (\textbf{Unit-I}) \qquad \frac{}{\Phi \mid \Theta \mid \Gamma \vdash \Diamond{:}\text{Unit}}\ (\top\textbf{-I})$$

We change the syntax "$\top$" to "Unit" and add the contexts Φ, Θ, Γ. We will do this for every rule which has empty contexts to subsume the weakening rules of the paper. The unit term always has the unit type as its type.

5.3.4. Variable lookup

We have three kinds of variables we can lookup. They are type variables, term variables and parameters. The paper already has rules for the type and term variables. We need to rewrite them. We add a new rule for looking up a parameter.

The rule

$$\frac{\vdash \Theta \quad \textbf{TyCtx} \qquad \vdash \Gamma \quad \textbf{Ctx}}{\Theta, X : \Gamma \multimap * \mid \emptyset \vdash X : \Gamma \multimap *}\ \textbf{TyVar-I}$$

gets rewritten to

$$\frac{\Theta(X) \rightsquigarrow \Gamma \multimap * \qquad \vdash \Gamma_1 \quad \textbf{Ctx}}{\Phi \mid \Theta \mid \Gamma_1 \vdash X : \Gamma \multimap *}\ \textbf{TyVar-I}$$

The rule

$$\frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}\ (\textbf{Proj})$$

gets rewritten to

$$\frac{\Gamma(x) \rightsquigarrow A}{\Phi \mid \Theta \mid \Gamma \vdash x : A} \textbf{ (Proj)}$$

The rule for looking something up in the parameter context is:

$$\frac{\Phi(X) \rightsquigarrow \Gamma \multimap * \qquad \vdash \Gamma_1 \quad \textbf{Ctx}}{\Phi \mid \Theta \mid \Gamma_1 \vdash X : \Gamma \multimap *} \textbf{ TyVar-I}$$

In the rule from the paper we can only infer the type or kind of the last variable in the context. In our rules we just look up the variable in the context. These rules can check the same thing if we take the weakening rules into account. With them we can just weaken the context until we get to the desired variable.

5.3.5. Type and expression instantiation

We can instantiate types and terms. The rule

$$\frac{\Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \multimap * \qquad \Gamma_1 \vdash t : B}{\Theta \mid \Gamma_1 \vdash A@t : \Gamma_2[t/x] \multimap *} \textbf{ (Ty-Inst)}$$

for instantiating types gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \multimap * \qquad \Phi \mid \Theta \mid \Gamma_1 \vdash t : B' \qquad B \equiv_\beta B'}{\Phi \mid \Theta \mid \Gamma_1 \vdash A@t : \Gamma_2[t/x] \multimap *} \textbf{ (Ty-Inst)}$$

For this rule we have to check if t has the expected type for the first variable in the context of A. In our version we just infer the type for A and t. Then we check if the first variable in the context is beta-equal to the type of t. If that isn't the case type checking fails. Otherwise we just substitute in the remaining context.

We also have a rule to instantiate terms. This rule

$$\frac{\Gamma_1 \vdash t : (x : A, \Gamma_2) \multimap B \qquad \Gamma_1 \vdash s : A}{\Gamma_1 \vdash t@s : \Gamma_2[s/x] \multimap B[s/x]} \textbf{ (Inst)}$$

gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma_1 \vdash t : (x : A, \Gamma_2) \multimap B \qquad \Phi \mid \Theta \mid \Gamma_1 \vdash s : A' \qquad A \equiv_\beta A'}{\Phi \mid \Theta \mid \Gamma_1 \vdash t@s : \Gamma_2[s/x] \multimap B[s/x]} \textbf{ (Inst)}$$

These rules are similar to the rule for type instantiation. Here we have to check(or infer) a term instead of a type. We also have to substitute s in the result type of t(in the case of types its always $*$, which obviously has no free variables).

5.3.6. Parameter abstraction

The rule

$$\frac{\Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \mathrel{-\!\!\triangleright} *}{\Theta \mid \Gamma_1 \vdash (x).B : (x : A, \Gamma_2) \mathrel{-\!\!\triangleright} *}\ \textbf{(Param-Abstr)}$$

gets rewritten to

$$\frac{\Phi| \ \Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \mathrel{-\!\!\triangleright} *}{\Phi| \ \Theta \mid \Gamma_1 \vdash (x : A).B : (x : A, \Gamma_2) \mathrel{-\!\!\triangleright} *}\ \textbf{(Param-Abstr)}$$

Here we just add the argument of the lambda to the expression context. Then we check the body of the lambda. In the syntax directed version we have to annotate the variable with its type, so we know which type we have to add to the context.

5.3.7. (co)inductive types

We have to separate the rule

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \qquad \Theta, X : \Gamma \mathrel{-\!\!\triangleright} * \mid \Gamma_k \vdash A_k : *}{\Theta \mid \emptyset \vdash \rho(X : \Gamma \mathrel{-\!\!\triangleright} *; \overrightarrow{\sigma}; \overrightarrow{A}) : \Gamma \mathrel{-\!\!\triangleright} *}\ \textbf{(FP-Ty)}$$

into multiple rules. First we need rules to check the definitions of (co)inductive types. These are

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \qquad \Phi| \ X : \Gamma \mathrel{-\!\!\triangleright} * \mid \Gamma_k \vdash A_k : * \qquad \vdash \phi \quad \textbf{ParCtx}}{\vdash \text{data X}\langle\Phi\rangle\ \Gamma \mathrel{-\!\!\triangleright} \text{Set where;}\ \overrightarrow{Constr_k : \Gamma_k \mathrel{-\!\!\triangleright} A_k \mathrel{-\!\!\triangleright} X\sigma_k}}\ \textbf{(FP-Ty)}$$

and

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \qquad \Phi| \ X : \Gamma \mathrel{-\!\!\triangleright} * \mid \Gamma_k \vdash A_k : * \qquad \vdash \phi \quad \textbf{ParCtx}}{\vdash \text{codata X}\langle\Phi\rangle : \Gamma \mathrel{-\!\!\triangleright} \text{Set where;}\ \overrightarrow{Destr_k : \Gamma_k \mathrel{-\!\!\triangleright} X\sigma_k \mathrel{-\!\!\triangleright} A_k}}\ \textbf{(FP-Ty)}$$

Because we only allow top level definitions of (co)inductive types our rules have empty contexts. We first have to check if σ_k is a context morphism from Γ_k to Γ. This basically means that the terms in σ_k are of the types in Γ, if we check them in Γ_k. After that we have to check if the $\overrightarrow{A}$ (the arguments where we can have a recursive occurrence) are of kind $*$. Because this is a top level definition the context ϕ is provided by the code. So we have to check if it is valid. We will now have to rewrite the rules for context morphism. Here we just add the parameter context to the rules of the paper.

$$\frac{}{\Phi \vdash () : \Gamma_1 \triangleright \emptyset} \qquad \frac{\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2 \qquad \Phi \mid \Gamma_1 \vdash t : A[\sigma]}{\Phi \vdash (\sigma, t) : \Gamma_1 \triangleright (\Gamma_2, x : A)}$$

We also need a rule for the cases in which we are using these defined variables. This is:

$$\frac{\Phi \mid \Theta \mid \Gamma' \vdash \vec{A} : \Gamma_i \rightarrowtriangle *}{\Phi \mid \Theta \mid \Gamma' \vdash X\langle \vec{A} \rangle : \Gamma[\vec{A}] \rightarrowtriangle *}$$

Here X is a data or codata definition. The parser can decide if a variable is a such a definition or a local definition. Because we are type checking on the abstract syntax tree we also know Γ and Φ′. Γ is just the context from the definition and Φ is the parameter context. Because we already typed checked this definition we just have to check if the types given for the parameters have the right kind. Then we substitute these parameters in its type. We will now give the rules for checking if a list of parameters matches a parameter context.

$$\frac{}{\Phi \mid \Theta \mid \Gamma \vdash () : ()} \qquad \frac{\Phi \mid \Theta \mid \Gamma \vdash A : \Gamma' \rightarrowtriangle * \qquad \Phi \mid \Theta \mid \Gamma \vdash \vec{A} : \Phi'[A/X]}{\Phi \mid \Theta \mid \Gamma \vdash A, \vec{A} : (X : \Gamma' \rightarrowtriangle *, \Phi')}$$

We just check every variable for the kinds in Φ′ one after the other. We also have to substitute the type into the context. Because kinds in a parameter context can depend on variables previously defined in this context.

5.3.8. Constructor and Destructor

The rule for constructors

$$\frac{\mu(X : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrowtriangle * \qquad 1 \leq k \leq |\vec{A}|}{\vdash \alpha_k^{\mu(X:\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : A_k[\mu/X]) \rightarrowtriangle \mu@\sigma_k}\ \textbf{(Ind-I)}$$

gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma \vdash \vec{B} : \Phi'}{\Phi \mid \Theta \mid \Gamma \vdash \mathrm{Constr}\langle \vec{B} \rangle : (\Gamma_k[\vec{B}], y : A_k[\mu/X][\vec{B}]) \rightarrowtriangle \mu@\sigma_k[\vec{B}]}\ \textbf{(Ind-I)}$$

The rule for destructors

$$\frac{\nu(X : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrowtriangle * \qquad 1 \leq k \leq |\vec{A}|}{\vdash \xi_k^{\nu(X;\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : \nu@\sigma_k) \rightarrowtriangle A_k[\nu/X]}\ \textbf{(Coind-E)}$$

gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma \vdash \vec{B} : \Phi'}{\Phi \mid \Theta \mid \Gamma \vdash \mathrm{Destr}\langle \vec{B} \rangle : (\Gamma_k[\vec{B}], y : \nu@\sigma_k)[\vec{B}] \rightarrowtriangle A_k[\nu/X][\vec{B}]}\ \textbf{(Ind-I)}$$

In the paper de/constructors are anonymous. They come together with their type. Therefor we have to check if this type is valid. Constructors construct their type. So their output value is their type μ applied to the context morphism σ_k, where k is the number of the constructor. They become as input the context Γ_k, which is implicit in the paper, and a value of type $A_k[\mu/X]$, which is the type, which can contain the recursive occurrence. Destructors are destructing their type so we get their type ν applied to σ_k as input and $A_k[\nu/X]$ as output.

In our rules, in contrast to the paper, the de/constructors refer to some type which we have already type checked. We just have to check the parameters. Every term we need is in the Haskell representation of the de/constructor. The de/constructor has the type which we have defined in the data definition. We just substitute the type itself for the free variable. At last we need to substitute the parameters for the respective variables.

5.3.9. Recursion and Corecursion

The rule

$$\frac{\vdash C : \Gamma \rightarrowtriangle * \qquad \Delta, \Gamma_k, y_k : A_k[C/X] \vdash g_k : (C@\sigma_k) \qquad \forall k = 1, \ldots, n}{\Delta \vdash \text{rec } \overrightarrow{(\Gamma_k, y_k).g_k} : (\Gamma, y : \mu@id_\Gamma) \rightarrowtriangle C@id_\Gamma} \ \textbf{(Ind-E)}$$

gets rewritten to

$$\frac{\vdash C : \Gamma \rightarrowtriangle * \qquad \begin{array}{c} \vdash \Gamma \equiv_\beta \Gamma'[\vec{D}] \\ \overrightarrow{\vdash B_k \equiv_\beta (C@\sigma_k[\vec{D}])} \end{array} \qquad \begin{array}{c} \Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi' \\ \overrightarrow{\Phi \,\|\, \Delta, \Gamma_k[\vec{D}], y_k : A_k[\vec{D}][C/X] \vdash g_k : B_k} \end{array}}{\Phi \mid \Theta \mid \Delta \vdash \text{rec } \mu\langle \vec{D} \rangle \text{ to C}; \overrightarrow{\text{Constr}_k \vec{x_k}\ y_k = g_k} : (\Gamma, y : \mu[\vec{D}]@id_\Gamma) \rightarrowtriangle C@id_\Gamma} \ \textbf{(Ind-E)}$$

We are recursing over some previously inductively defined type μ to some type C. This types must have the same context. Recursing is done by listing each constructor with the result, which the whole expression should have if we apply it to this constructor. This result can refer to the arguments of the constructor via the variables $\vec{x_k}, y_k$. The type must be the result type C applied to the σ_k of this constructor. In the syntax directed version we also have to check the parameters. We check if the types match by inferring them and compare them on beta equality.

We have a similar rule for corecursion. It

$$\frac{\vdash C : \Gamma \rightarrowtriangle * \qquad \Delta, \Gamma_k, y_k : (C@\sigma_k) \vdash g_k : A_k[C/X] \qquad \forall k = 1, \ldots, n}{\Delta \vdash \text{corec } \overrightarrow{(\Gamma_k, y_k).g_k} : (\Gamma, y : C@id_\Gamma) \rightarrowtriangle \nu@id_\Gamma} \ \textbf{(Coind-I)}$$

gets rewritten to

$$\frac{\vdash C : \Gamma \rightarrowtriangle * \qquad \begin{array}{c} \vdash \Gamma \equiv_\beta \Gamma'[\vec{D}] \\ \overrightarrow{\vdash B_k \equiv_\beta A_k[\vec{D}][C/X]} \end{array} \qquad \begin{array}{c} \Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi' \\ \overrightarrow{\Phi \,\|\, \Delta, \Gamma_k[\vec{D}], y_k : (C@\sigma_k[\vec{D}]) \vdash g_k : B_k} \end{array}}{\Phi \mid \Theta \mid \Delta \vdash \text{corec C to } \nu\langle \vec{D} \rangle; \overrightarrow{\text{Destr}_k \vec{x_k}\ y_k = g_k} : (\Gamma, y : C@id_\Gamma) \rightarrowtriangle \nu[\vec{D}]@id_\Gamma} \ \textbf{(Coind-I)}$$

A corecursion produces a coinductive type ν. We have to give it a type C and list the destructors together with the expression they should be destructed to. We get the syntax directed rule analog as in the case of recursion.

5.4. Evaluation

There are two kinds of reduction steps in this system. The implementation of this is in `Eval.hs`. Will give the formal definition in the following.

The first is a reduction on the type level, $\longrightarrow$. It is defined like follows:

$$((x).A)@t \longrightarrow_p A[t/x]$$

It is standard beta reduction. If we apply a lambda $(x).A)$ to a term t we substitute this term for the binding variable x in the body. This body is then the result of the reduction.

The other is the reduction on the term level, $\succ$. To define this reduction we need a action on types (written $\widehat{C}(A)$) and terms (written $\widehat{C}(t)$), where the following holds.

$$\frac{X : \Gamma_1 \mathrel{-\!\triangleright} * \mid \Gamma_2' \vdash C : \Gamma_2 \mathrel{-\!\triangleright} * \qquad \Gamma_1, x : A \vdash t : B}{\Gamma_2', \Gamma_2, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

Here we have a type C with a free type variable X and a term t of type B with a free term variable x of type A. If we use the action of this type on t we get a term with a type of this action on B. This term contains a free term variable x of type, the action applied to A. The type action is implemented in the module `TypeAction.hs`. Both the type action and the evaluation are done in the `Eval` monad. This monad has access to the previously defined declarations. We will now define the type action.

Definition 1. *Let $n \in \mathbb{N}$ and $1 \leq i \leq n$. Let:*

$$\begin{aligned} X_1 : \Gamma_1 \mathrel{-\!\triangleright} *, \ldots, X_n : \Gamma_n \mathrel{-\!\triangleright} * \ \mid \ \Gamma' \vdash C : \Gamma \mathrel{-\!\triangleright} * \\ \Gamma_i \vdash A_i : * \\ \Gamma_i \vdash B_i : * \\ \Gamma_i, x : A_i \vdash t_i : B_i \end{aligned}$$

Then we define the type action on terms inductively over C

$$
\begin{array}{ll}
\widehat{C}(\vec{t}, t_{n+1}) = \widehat{C}(\vec{t}) & \textit{for } \textbf{\textit{(TyVarWeak)}} \\
\widehat{X_i}(\vec{t}) = t_i & \\
\widehat{C'@s}(\vec{t}) = \widehat{C'}(\vec{t})[s/y], & \textit{for } \Theta \mid \Gamma' \vdash C' : (y,\Gamma) \to * \\
\widehat{(y).C'}(\vec{t}) = \widehat{C'}(\vec{t}), & \textit{for } \Theta \mid (\Gamma', y) \vdash C' : \Gamma \to * \\
\widehat{\mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D})} = rec^{R_A}\overrightarrow{(\Delta_k, x).g_k}@id_\Gamma@x & \textit{for } \Theta, Y : \Gamma \to * \mid \Delta_k \vdash D_k : * \\
\quad \textit{with } g_k = \alpha_k^{R_B}@id_{\Delta_k}@\left(\widehat{D_k}(\vec{t}, x)\right) & \\
\quad \textit{and } R_A = \mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}]) & \\
\quad \textit{and } R_B = \mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}]) & \\
\widehat{\nu(Y : \Gamma \to *; \vec{\sigma}; \vec{D})} = corec^{R_B}\overrightarrow{(\Delta_k, x).g_k}@id_\Gamma@x & \textit{for } \Theta, Y : \Gamma \to * \mid \Delta_k \vdash D_k : * \\
\quad \textit{with } g_k = \widehat{D_k}(\vec{t}, x)[(\xi_k^{R_A}@id_{\Delta_k}@x)/x] & \\
\quad \textit{and } R_A = \mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}]) & \\
\quad \textit{and } R_B = \mu(Y : \Gamma \to *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}]) &
\end{array}
$$

And the type action on types as follow

$$\widehat{C}(\vec{A}) = C[\overrightarrow{(\Gamma_i).A}/\vec{X}]@id_\Gamma$$

The type action generates a term with a free variable x. In the type of this term we have changed all the free variables to the types of $\vec{t}$. We will show the proof in appendix A.

The reduction on terms is subdivided into an reduction on recursion and one on corecursion. Here $\sigma_k \bullet \tau$ is a context morphism, where we first substitute with τ and then with σ_k.

The reduction on recursion is defined as follows

$$\mathrm{rec}\overrightarrow{(\Gamma_k, y_k).g_k}@(\sigma_k \bullet \tau)@(\alpha_k@\tau@u) \succ g_k\left[\widehat{A_k}(\mathrm{rec}\overrightarrow{(\Gamma_k, y_k).g_k}@\mathrm{id}_\Gamma@x)/y_k\right][\tau, u]$$

If we apply a recursion $\mathrm{rec}\overrightarrow{(\Gamma_k, y_k).g_k}$ to this context morphism and a constructor $\alpha_k@\tau@u$, which is fully applied, we lookup the case for this constructor. In this case we substitute τ for the variables from Γ_k and u, where we apply the recursion to all recursive occurrences, for y_k. For this application we need the type action. So a recursion is destructing an inductive type and all its recursive occurrences to another type, while we use different cases for the different constructors of the type.

On the contrary corecursion is constructing a coinductive type. It is defined like follows:

$$\xi_k@\tau@(\mathrm{corec}\overrightarrow{(\Gamma_k, y_k).g_k}@(\sigma_k \bullet \tau)@u) \succ \widehat{A_k}(\mathrm{corec}\overrightarrow{(\Gamma_k, y_k).g_k}@\mathrm{id}_\Gamma@x)[g_k/x][\tau, u]$$

If we apply a destructor together with its arguments for it context ,$\xi_k @ \tau$, on such a construction $(\mathrm{corec}\overrightarrow{(\Gamma_k, y_k)}.g_k @ (\sigma_k \bullet \tau) @ u)$, we are taking the case of this destructor. In this case we are applying the corecursion to all recursive occurrences. τ and u are substituted as in recursion.

6. Examples

In this section we reiterate the example types from the paper. We use our syntax, which is defined in 5.1. We will also show some functions on these types. On some of them we will show the reduction steps in detail.

6.1. Terminal and Initial Object

The terminal object is a type which has exactly one value. In category theory every object in the category has an unique morphism to it. We define it as a coinductive type `Terminal` with no destructors . It gets a terminal and returns a terminal. To get a terminal value we use corecursion on the unit type, which is the first class terminal object.

```
codata Terminal : Set where
terminal = corec Unit to Terminal where @ ◊
```

Contrary to the definition in the paper there is no destructor `Terminal`. In the paper definitions of coinductive or inductive types need at least one de/constructor. Therefore our definition wouldn't work.

The initial object is a type which has no values. In category theory it is the object which has an unique morphism to every other object in the category. We define it inductively as `Intial` with no constructor. In the paper it is defined with one constructor. This constructor want's one value of the same type. We can't have a value of this type, because to get one we already need one. Our way of defining it is shorter and more clear. We can't construct an value of this type because we have no constructors. If we could get something of type `Intial`, we could generate with `exfalsum` a value of arbitrary type `C`.

```
data Initial : Set where
exfalsum⟨C : Set⟩ = rec Initial to C where
```

6.2. Natural Numbers and Extended Naturals

We use the classical peano numbers to define natural numbers. Therefor we use the inductive type `Nat` with the constructors `Zero` and `Suc`. `Zero` is just the number zero. Every constructor has to have an argument, which can contain a recursive occurrence. Every Type `A` is isomorphic to the function type `Terminal → A`. So we use `Terminal` for this occurrence. `Suc` is the successor. So the meaning of `Suc n` is $n+1$.

```
data Nat : Set where
   Zero : Terminal → Nat
   Suc : Nat → Nat
zero = Zero @ ◊
one = Suc @ zero
```

We can then define a identity recursion on it to see how reduction works. It's a recursion which goes from a `Nat` to `Nat` and gives back in every case its input.

```
id = rec Nat to Nat where
       Zero u = Zero @ u
       Succ n = Succ @ n
```

We use it on one to see all cases.

```
id @ one = id @ (Succ @ zero)
         > Succ @ n[X̂(id @ x)/n]  [zero]
         = Succ @ X̂(id @ x)  [zero]
         = Succ @ (id @ x)[zero]
         = Succ @ (id @ zero)
         = Succ @ (id @ (Zero @ ◊))
         > Succ @ (Zero @ u[Ûnit(id @ x)/u][◊])
         = Succ @ (Zero @ u[Ûnit(id @ x)/u][◊])
         = Succ @ (Zero @ Ûnit(id @ x)[◊])
         = Succ @ (Zero @ x)[◊]
         = Succ @ (Zero @ x) = Succ @ zero = one
```

As expected the identity recursion applied to one gives back one.

We will now define extended naturals. There are also called co-natural numbers. There are natural numbers with an additional value, infinity. We define it coinductively with the predecessor as its only destructor. The predecessor is either not defined or another natural number. We use the type `Maybe` to describe something which is either present (the constructor `Just`) or absent(the constructor `Nothing`). We can define the successor as a corecursion. The predecessor of the successor of `x` is just `x`. So the only case of `corec` returns a `Just x` (remember `Prec` returns a `Maybe⟨ Conat⟩` not a `Conat`).

```
data Maybe⟨A : Set⟩ : Set where
  Nothing : Unit → Maybe
  Just : A → Maybe
nothing⟨A⟩ = Nothing⟩A⟩ @ ◊
codata Conat : Set where
  Prec : Conat → Maybe⟨Conat⟩
succ = corec Conat to Conat where
         Prec x = Just⟨Conat⟩ @ x
```

We now define the values zero and infinity

```
zero = (corec Unit to Conat where
          {Prev x = nothing⟨Unit⟩}) @ ◊
```

```
infinity = (corec Unit to Conat where
             {Prev x = Just⟨Conat⟩ @ x}) @ ◊
```

For `zero` the predecessor is absent, there is no predecessor of 0 in the natural numbers, so we give pack `Nothing`. We then have to apply the `corec` to ◊ to get the value. The predecessor of `infinity` should also be `infinity`. We apply the `corec` to another `Conat`, so the `x` is also a `Conat`. We will know see that the predecessor on this values give back the right value.

$$
\begin{aligned}
\mathrm{Prev} \mathbin{@} \mathrm{zero} &\succ \widehat{\mathrm{Maybe}}\langle X\rangle \left(\underbrace{\substack{\text{corec Unit to Conat where}\\ \{\,\mathrm{Prev}\ x = \mathrm{nothing}\langle\mathrm{Unit}\rangle\,\}}}_{t_1} \mathbin{@} x \right)[\mathrm{nothing}\langle\mathrm{Unit}\rangle/x][\lozenge] \\
&= \text{rec } \mathrm{Maybe}\langle\mathrm{Unit}\rangle \text{ to } \mathrm{Maybe}\langle\mathrm{Conat}\rangle \text{ where} \\
&\qquad \{\mathrm{Nothing}\ u = \mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} \widehat{\mathrm{Unit}}(t_1, u) \\
&\qquad\ \ \mathrm{Just}\ c = \mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} \widehat{X}(t_1, c)\} \mathbin{@} x[\mathrm{nothing}\langle\mathrm{Unit}\rangle/x][\lozenge] \\
&= \underbrace{\substack{\text{rec Maybe}\langle\text{Unit}\rangle \text{ to Maybe}\langle\text{Conat}\rangle \text{ where}\\ \{\,\mathrm{Nothing}\ u=\mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} u\\ \mathrm{Just}\ c=\mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} t_1\}}}_{t_2} \mathbin{@} \mathrm{nothing}\langle\mathrm{Unit}\rangle \\
&\succ \mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} u[\widehat{\mathrm{Unit}}(t_2 \mathbin{@} x)/u][\lozenge] \\
&= \mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} u[x/u][\lozenge] \\
&= \mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} \lozenge
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Prev} \mathbin{@} \mathrm{infinity} &\succ \widehat{\mathrm{Maybe}}\langle X\rangle \left(\underbrace{\substack{\text{corec Unit to Conat where}\\ \{\,\mathrm{Prev}\ x = \mathrm{Just}\langle\mathrm{Unit}\rangle \mathbin{@} x\,\}}}_{t_1} \mathbin{@} x \right)[\mathrm{Just}\langle\mathrm{Unit}\rangle \mathbin{@} /x][\lozenge] \\
&= \text{rec } \mathrm{Maybe}\langle\mathrm{Unit}\rangle \text{ to } \mathrm{Maybe}\langle\mathrm{Conat}\rangle \text{ where} \\
&\qquad \{\ \mathrm{Nothing}\ u = \mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} \widehat{\mathrm{Unit}}(t_1, u) \\
&\qquad\ \ \mathrm{Just}\ c = \mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} \widehat{X}(t_1, c)\ \} \mathbin{@} x[\mathrm{Just}\langle\mathrm{Unit}\rangle \mathbin{@} /x][\lozenge] \\
&= \underbrace{\substack{\text{rec Maybe}\langle\text{Unit}\rangle \text{ to Maybe}\langle\text{Conat}\rangle \text{ where}\\ \{\,\mathrm{Nothing}\ u=\mathrm{Nothing}\langle\mathrm{Conat}\rangle \mathbin{@} u\\ \mathrm{Just}\ c=\mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} t_1\,\}}}_{t_2} \mathbin{@} \mathrm{Just}\langle\mathrm{Unit}\rangle \mathbin{@} \\
&\succ \mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} t_1[\widehat{\mathrm{Unit}}(t_2 \mathbin{@} x)/x][\lozenge] \\
&= \mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} t_1[x/x][\lozenge] \\
&= \mathrm{Just}\langle\mathrm{Conat}\rangle \mathbin{@} \mathrm{infinity}
\end{aligned}
$$

6.3. Binary Product and Coproduct

The product is defined as a coinductive type. It has two destructors. The first gives back the first element. And the second the second. To use this type, the types A and B have to be instantiated to concrete types. We don't have type polymorphism in our language. We also define a pair expression which generates a pair over corecursion.

```
codata Product⟨A : Set, B : Set⟩ : Set where
   Fst : Product ⇀ A
   Snd : Product ⇀ B
pair⟨A : Set, B : Set⟩ (x:A, y:B) = corec Unit where
                                       { Fst u ⇀ x
                                       ; Snd u ⇀ y} @ ◊
```

For types with other contexts we have to define different Products. For example if `B` depends on `Nat`, we define product like the following:

```
codata Pair⟨A : Set, B : (n : Nat) ⇀ Set⟩ : (n : Nat) ⇀ Set where
  First : (n : Nat) ⇀ Pair n ⇀ A
  Second : (n : Nat) ⇀ Pair n ⇀ B @ n
```

Here the product also depends on `Nat`. If `A` or `B` depends on values the product must also depend on these values. This is the product, which is used for the definition of vectors in [BG16].

On `Product` we can define the swap function.

```
swap⟨A : Set, B : Set⟩ =
  corec Product⟨A,B⟩ to Product⟨B,A⟩ where
         Fst x ⇀ Snd x
         Snd x ⇀ Fst x
```

This is a well typed function as shown by the following proof

$$\frac{(A:*,B:*) \Vdash \text{Product}\langle A,B\rangle : * \qquad \begin{array}{r}(A:*,B:*) \,\|\, (x:A) \vdash \text{Snd @ x} : \text{Product}\langle A,B\rangle \text{ ⓐ} \\ (A:*,B:*) \,\|\, (y:B) \vdash \text{Fst @ y} : \text{Product}\langle A,B\rangle \text{ ⓑ}\end{array}}{(A:*,B:*) \Vdash \text{swap} : (\text{p} : \text{Product}\langle A,B\rangle) \rightharpoonup \text{Product}\langle B,A\rangle}$$

We show ⓐ in the following proof. ⓑ works analog.

$$\frac{(A:*,B:*) \,\|\, (x:A) \vdash \text{Snd} : (x:A) \rightharpoonup \text{Product}\langle A,B\rangle \qquad \dfrac{(x:A)(x) \rightsquigarrow A}{(x:A) \vdash x : A}}{(A:*,B:*) \,\|\, (x:A) \vdash \text{Snd @ x} : \text{Product}\langle A,B\rangle}$$

For brevity we omitted the beta equality premises and the checking for of the parameters. The beta equality premises wouldn't be interesting because they all already syntactically identical.

The Binary Coproduct corresponds to the Either type in Haskell. It is defined as an inductive type. It is either `A` or `B`. We have one constructor `Left` for `A` and one constructor Right for `B`.

```
data Coproduct⟨A,B⟩ : Set where
   Left : A ⇀ Coproduct
   Right : B ⇀ Coproduct
```

6.4. Sigma and Pi Type

The sigma type is a dependent pair of two types. The second type can depend on the value of the first type. It corresponds to exists in logic. We define it as an inductive type and call the constructor `Exists`.

```
data Sigma⟨A : Set,B : (x : A) → Set⟩ : Set where
    Exists : (x:A) → B x → Sigma
```

The pi type is a generalization of the function type to dependent types. The type of the codomain or result of a function can depend on the value We define it as a coinductive type. To destruct a function we just apply it to a value. So the destructor is `Apply`.

```
codata Pi⟨A : Set,B : (x : A) → Set⟩ : Set where
    Apply : (x : A) → Pi x → B
```

To construct a function we use corecursion on `Unit`. The identity function is defined like this

```
id⟨A : Set⟩ = corec Unit to Pi⟨A,(v:A).A⟩ where
         { Apply v p = v } @ ◊
```

Evaluation on one goes as follows:

```
apply = Apply⟨Nat,(v : Nat).Nat⟩
one = S @ (Z @ )
apply @ id⟨Nat⟩ @ one
= apply @ one @ ((corec Unit to Pi⟨Nat,(x:Nat).Nat⟩ where
                          Apply v p = v ) @ ◊)
```

$$\succ \widehat{\mathrm{Nat}}\left(\underbrace{\begin{array}{c}\text{corec Unit to Pi where}\\ \text{\{Apply' v \_ = v\} @ x}\end{array}}_{t}\right)[\mathrm{v/x}][\mathrm{one},\lozenge]$$

```
= (rec Nat to Nat where
      Zero x = Zero @ (Unit^(t,x))
      Succ x = Suc @ (Y^(t,x)))@x[v/x][one,◊]
= (rec Nat to Nat where
      Zero x = Zero @ (Unit^(t))
      Succ x = Suc @ x)@x[v/x][one,◊]
= (rec Nat to Nat where
      Zero x = Zero @ (Unit^())
      Succ x = Suc @ x) @ x[v/x][one,◊]
= (rec Nat to Nat where
      Zero x = Zero @ x
      Succ x = Suc @ x) @ x[v/x][one,◊]
= (rec Nat to Nat where
      Zero x = Zero @ x
      Succ x = Suc @ x) @ v[one,◊]
= (rec Nat to Nat where
      Zero x = Zero @ x
      Succ x = Suc @ x) @ one
= one
```

6.5. Vectors and Streams

Vectors are a standard example for dependent types. They are like lists, except their type depends on their length. For example a vector `[1;2]` has type `Vector⟨ Nat⟩ 2`, because its length is 2. It has 2 constructors `Nil` and `Cons` like lists. `Nil` gives back the empty vector. Because the length of the empty vector is zero its return type is `Vector 0`. The second constructor `Cons` takes a natural number `k`, a value of type `A` and a vector of length `k`, a `Vector k`. It returns a new vector. Its head is the first argument and its tail the second. So the results length is one more then the second argument . Therefore it is `Vector (Suc k)`. In [BG16] the head and tail are encoded in a pair.

```
data Vector⟨A : Set⟩ : (n:Nat) ⇾ Set where
  Nil : Unit ⇾ Vector zero
  Cons : (k:Nat, v:A) ⇾ Vector @ k ⇾ Vector (Suc @ k)
nil⟨A : Set⟩ = Nil⟨A : Set⟩ @ ◊
```

The function `extend` takes a value `x` and extends it to a vector.

```
extend⟨A : Set⟩ =
  rec Vec⟨A⟩ to ((x).Vec⟨ A⟩ @ (Suc x) where
    Nil u = Cons⟨A⟩ @ x @ nil⟨A⟩
    Cons k v = Cons⟨A⟩ @ (Suc @ k) @ v
```

The type checking of this function goes as follows:

$$\frac{\begin{array}{c}(\mathrm{A} : \mathrm{Set})\|\!\vdash (\mathrm{x}).(\mathrm{Vec}\langle \mathrm{A}\rangle \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{x})) : (\mathrm{k{:}\ Nat}) \multimap * \\ (\mathrm{A{:}\ Set})\|(\mathrm{u} : \mathrm{A}) \vdash \mathrm{Cons}\langle \mathrm{A}\rangle \mathbin{@} 0 \mathbin{@} (\mathrm{Nil}\langle \mathrm{A}\langle \mathbin{@}) : (\mathrm{x}).(\mathrm{Vec}\langle \mathrm{A}\rangle \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{x})) \mathbin{@} 0 \\ (\mathrm{k} : \mathrm{Nat},\ \mathrm{v} : (\mathrm{x}).(\mathrm{Vec} \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{x})) \mathbin{@} \mathrm{k}) \vdash \mathrm{Cons}\rangle \mathrm{A}\langle \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{k}) \mathbin{@} \mathrm{v} : (\mathrm{x}).(\mathrm{Vec} \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{x})) \mathbin{@} (\mathrm{Suc} \mathbin{@} \mathrm{k})\end{array}}{\vdash \mathrm{extend}\langle \mathrm{A}\rangle : (\mathrm{k{:}Nat,y} : \mathrm{Vec}\langle \mathrm{A}\rangle \mathbin{@} \mathrm{k}) \multimap (\mathrm{x}).(\mathrm{Vec}\langle \mathrm{A}\rangle \mathbin{@} (\mathrm{Suc\ x})) \mathbin{@} \mathrm{k}}$$

As an example we evaluate a vector of length 1 with this function. We choose length one to see all `rec` cases.

$$\begin{array}{l}
\mathrm{extend}\langle \mathrm{Nat}\rangle @ 1 @ (\mathrm{Cons}\langle \mathrm{Nat}\rangle @ 0 @ 0 @ \mathrm{nil}\langle \mathrm{Nat}\rangle) \\
= \mathrm{extend}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k \bullet 0) @ (\mathrm{Cons}\langle \mathrm{Nat}\rangle @ 0 @ 0 @ \mathrm{nil}\langle \mathrm{Nat}\rangle) \\
\succ \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k) @ v \left[\widehat{X @ k}(\mathrm{extend}\langle \mathrm{Nat}\rangle @ n @ x)/v\right][0, \mathrm{nil}\langle \mathrm{Nat}\rangle] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k) @ v \left[\widehat{X}(\mathrm{extend}\langle \mathrm{Nat}\rangle @ n @ x)[k/n]/v\right][0, \mathrm{nil}\langle \mathrm{Nat}\rangle] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k) @ v [\mathrm{extend} @ n @ x[k/n]/v][0, \mathrm{nil}\langle \mathrm{Nat}\rangle] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k) @ v [\mathrm{extend} @ k @ x/v][0, \mathrm{nil}\langle \mathrm{Nat}\rangle] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ k) @ (\mathrm{extend} @ k @ x)[0, \mathrm{nil}\langle \mathrm{Nat}\rangle] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ (\mathrm{Suc} @ 0) @ (\mathrm{extend} @ 0 @ (\mathrm{nil}\langle \mathrm{Nat}\rangle)) \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ 1 @ (\mathrm{extend} @ 0 @ (\mathrm{Nil}\langle \mathrm{Nat}\rangle @)) \\
\succ \mathrm{Cons}\langle \mathrm{Nat}\rangle @ 1 @ (\mathrm{Cons}\langle \mathrm{Nat}\rangle @ 0 @ (\mathrm{Nil}\langle \mathrm{Nat}\rangle @)) \left[\widehat{\mathrm{Unit}}(\mathrm{extend} @ k @ x)/u\right][\Diamond] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ 1 @ (\mathrm{Cons}\langle \mathrm{Nat}\rangle @ 0 @ (\mathrm{Nil}\langle \mathrm{Nat}\rangle @ x))[\Diamond] \\
= \mathrm{Cons}\langle \mathrm{Nat}\rangle @ 1 @ (\mathrm{Cons}\langle \mathrm{Nat}\rangle @ 0 @ (\mathrm{Nil}\langle \mathrm{Nat}\rangle @))
\end{array}$$

Here we write 1 for `Suc @ (Zero @ )` and 0 for `Zero @ ◊`.

With the help of extended naturals, we can define partial streams. These are streams which depend on there definition depth. Like non-dependent streams they are coinductive and have 2 destructors for head and tail.

```
codata PStr⟨A : Set⟩: (n: ExNat) → Set where
    hd : (k : ExNat) → PStr⟨A⟩ (succE k) → A
    tl : (k : ExNat) → PStr⟨A⟩ (succE k) → PStr⟨A⟩ @ k
```

These streams are like vectors except they also can be infinite long. This is in contrary to non dependent streams. A non dependent stream could not be of length zero. Because then a call of `hd` and `tl` on it wouldn't be defined. In the dependent case the type checker wouldn't allow such a call because `hd` and `tl` expect streams which are at least of length one. We can then define `repeat`.

```
repeat⟨A : Set⟩(x : A, n : Conat) =
  corec (n : Conat).Unit to PStr⟨A⟩ where
    { Hd k s = x
    ; Tl k s = ◊ } @ n @ ◊
```

This function gets a value and an extended natural number. It generate an constant partial stream of that value with the number as its length.

7. Conclusion

We have implemented a depend type theory with inductive and coinductive types. In this theory, contrary to coq and agda, coinductive types can also depend on values. Contrary to the theory of the paper we can define schemata like `Maybe⟨ A : Set⟩` where `A` can be an arbitrary type of kind `Set`.

One downside is that we don't have universes. This prevents type polymorphism. Further work needs do be done to solve this. Another problem is, that each constructor or destructor has at least one argument. The argument with the recursive occurrence. For example we have to apply an unit to the constructors of a boolean type. We could allow recursive occurrences in the contexts of the constructors and destructors. This makes it possible to remove the argument with the recursive occurrence. We then have to change the evaluation rules.

Our system allowed us to define the (depended) function type. Therefor we don't have it as primitive expression. We are hopeful, that in the future we get an more mainstream language, like Coq or Agda, where the depended function is definable. As already mentioned in the introduction this would lead to a symmetrical language.

A. Type action proof

Theorem 1. $(\Gamma).A@id_\Gamma \leftrightarrow_T A$

Proof. We show this by induction on the length of Γ

- $\Gamma = \epsilon$:
$$A \longleftrightarrow_T A$$

- $\Gamma = x : B, \Gamma'$:
$$(x : B, \Gamma').A@x@\mathrm{id}_{\Gamma'} \longrightarrow_p (\Gamma').A@\mathrm{id}_{\Gamma'}[x/x] = (\Gamma').A@\mathrm{id}_{\Gamma'} \overset{IdH.}{\longleftrightarrow}_T A$$

$\square$

Theorem 2. *The following rule holds*

$$\frac{x : A \vdash t : B \qquad A \longleftrightarrow_T A'}{x : A' \vdash t : B}$$

Proof. We show this by induction on t $\square$

Theorem 3. *The typing rule (5) in the paper holds*

$$\frac{X : \Gamma_1 \multimap * \mid \Gamma' \vdash C : \Gamma \multimap * \qquad \Gamma_1, x : A \vdash t : B}{\Gamma', \Gamma, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

Proof. First we will generalize the rule to

$$\frac{X_1 : \Gamma_1 \multimap *, \ldots, X_n : \Gamma_n \multimap * \mid \Gamma' \vdash C : \Gamma \multimap * \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})}$$

Then we gonna show it by Induction on the derivation $\mathcal{D}$ of C

- $\mathcal{D} = \dfrac{}{\vdash \top : *}$ **($\top$-I)**

  Then the type actions got calculated as follows
$$\widehat{\top}(\vec{A}) = \widehat{\top}() = \top$$
$$\widehat{\top}(\vec{t}) = \widehat{\top}() = x$$
$$\widehat{\top}(\vec{B}) = \widehat{\top}() = \top$$

  We than got the following prooftree

$$\dfrac{\vdash \top : *}{x : \top \vdash x : \top}\ \textbf{(Proj)}$$

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_{n-1} : \Gamma_{n-1}\ \textbf{TyCtx}} \qquad \overset{\mathcal{D}_2}{\Gamma_n\ \textbf{Ctx}}}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \rightarrowtriangle * \mid \emptyset \vdash X_n : \Gamma_n \rightarrowtriangle *}\ \textbf{TyVar-I}$

  Again we calculate the type actions

$$\widehat{X_n}(\vec{A}) = X_n[\overrightarrow{(\Gamma_i).A}/\vec{X}]@\mathrm{id}_{\Gamma_n} = X_n[(\Gamma_n).A_n/X_n]@\mathrm{id}_{\Gamma_n} = (\Gamma_n).A_n@\mathrm{id}_{\Gamma_n}$$
$$\widehat{X_n}(\vec{t}) = t_n$$
$$\widehat{X_n}(\vec{B}) = X_n[\overrightarrow{(\Gamma_i).B}/\vec{X}]@\mathrm{id}_{\Gamma_n} = X_n[(\Gamma_n).B_n/X_n]@\mathrm{id}_{\Gamma_n} = (\Gamma_n).B_n@\mathrm{id}_{\Gamma_n}$$

  We know from the first premise that $\Gamma = \Gamma_n$ and $\Gamma' = \emptyset$

  Here we got the prooftree

$$\dfrac{\dfrac{\Gamma_n, x : A \vdash t : B \qquad \dfrac{}{A \longleftrightarrow_T (\Gamma_n).A@\mathrm{id}_{\Gamma_n}}\ \text{Thrm. 1}}{\Gamma_n, x : (\Gamma_n).A@\mathrm{id}_{\Gamma_n} \vdash t : B}\ \text{Thrm. 2} \qquad \dfrac{}{B \longleftrightarrow_T (\Gamma_n).B@\mathrm{id}_{\Gamma_n}}\ \text{Thrm. 1}}{\Gamma_n, x : (\Gamma_n).A@\mathrm{id}_{\Gamma_n} \vdash t_n : (\Gamma_n).B@\mathrm{id}_{\Gamma_1}}\ \text{Conv}$$

- $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *} \qquad \overset{\mathcal{D}_2}{\Gamma_n\ \textbf{Ctx}}}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_{n+1} : \Gamma_{n+1} \rightarrowtriangle * \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *}\ \textbf{(TyVar-Weak)}$

  Here we got the prooftree

$$\dfrac{\dfrac{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_{n+1} : \Gamma_{n+1} \rightarrowtriangle * \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *}\ (*) \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', \Gamma, x : \underbrace{\widehat{C}(\vec{A})}_{\overset{(**)}{=}\widehat{C}(\vec{A}, A_{n+1})} \vdash \underbrace{\widehat{C}(\vec{t})}_{\overset{(***)}{=}\widehat{C}(\vec{t}, t_{n+1})} : \underbrace{\widehat{C}(\vec{B})}_{\overset{(**)}{=}\widehat{C}(\vec{B}, B_{n+1})}}\ \text{IdH.}$$

  (*) Here we undo **(TyVar-Weak)**

  (**) X_{n+1} doesn't occur free in C, otherwise $\mathcal{D}_1$ wouldn't be possible

  (***) Case for **(TyVar-Weak)** of type actions on terms

- $\mathcal{D} =$

$$\dfrac{\overset{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *} \qquad \overset{\mathcal{D}_2}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma', y : D \vdash C : \Gamma \rightarrowtriangle *}\ \textbf{(Ty-Weak)}$$

  Here we got the prooftree

$$\dfrac{\dfrac{\dfrac{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma', y : D \vdash C : \Gamma \rightarrowtriangle *}{X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma' \vdash C : \Gamma \rightarrowtriangle *}\ (*) \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})}\ \text{IdH.} \qquad X_1 : \Gamma_1 \rightarrowtriangle *, \ldots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) y \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})}\ \textbf{(Term-Weak)}$$

(*) Here we undo **(Ty-Weak)**

- $\mathcal{D} = \dfrac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrowtriangle * \qquad \Gamma' \vdash s : D}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C'@s : \Gamma \rightarrowtriangle *}$ **(Ty-Inst)**

Then we got the following induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrowtriangle * \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', y : D, \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}\,) : \widehat{C'}(\vec{B})}$$

Calculated type actions:

$$\begin{aligned}
&\widehat{C'@s}(\vec{A}) = C'@s[\overrightarrow{(\Gamma_i).A}/\vec{X}]@\mathrm{id}_\Gamma = C'[\overrightarrow{(\Gamma_i).A}/\vec{X}]@s@\mathrm{id}_\Gamma = \widehat{C'}(\vec{A})[s/y]\\
&\widehat{C'@s}(\vec{t}\,) = \widehat{C'}(\vec{t}\,)[s/y]\\
&\widehat{C'@s}(\vec{B}) = C'@s[\overrightarrow{(\Gamma_i).B}/\vec{X}]@\mathrm{id}_\Gamma = C'[\overrightarrow{(\Gamma_i).B}/\vec{X}]@s@\mathrm{id}_\Gamma = \widehat{C'}(\vec{B})[s/y]
\end{aligned}$$

We then got the following prooftree

$$\dfrac{\dfrac{\dfrac{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n \rightarrowtriangle * \mid \Gamma_2' \vdash C'@s : \Gamma_2[s/y] \rightarrowtriangle *}{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma_2' \vdash C' : (y : D, \Gamma_2) \rightarrowtriangle *}\,(*) \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma_2', y : D, \Gamma_2, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}\,) : \widehat{C'}(\vec{B})}\,\text{IdH.}}{\Gamma_2', \Gamma_2[s/y], x : \widehat{C'}(\vec{A})[s/y] \vdash \widehat{C'}(\vec{t}\,)[s/y] : \widehat{C'}(\vec{B})[s/y]}$$

(*) This is the reverse of **(Ty-Inst)**.

- $\mathcal{D} = \dfrac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma', y : D \vdash C' : \Gamma \rightarrowtriangle *}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash (y).C' : (y : D, \Gamma) \rightarrowtriangle *}$ **(Param-Abstr)**

Calculated type actions:

$$\begin{aligned}
\widehat{(y).C'}(\vec{A}) &= (y).C'[\overrightarrow{(\Gamma_i.A)}/\vec{X}]@\mathrm{id}_\Gamma\\
&= (y).(C'[\overrightarrow{(\Gamma_i.A)}/\vec{X}])@y@\mathrm{id}_\Gamma\\
&\longleftrightarrow_T (C'[\overrightarrow{(\Gamma_i.A)}/\vec{X}])@\mathrm{id}_\Gamma\\
&= \widehat{C'}(\vec{A})\\
\widehat{(y).C'}(\vec{t}\,) &= \widehat{C'}(\vec{t}\,)\\
\widehat{(y).C'}(\vec{B}) &= (y).C'[\overrightarrow{(\Gamma_i.B)}/\vec{X}]@\mathrm{id}_\Gamma\\
&= (y).(C'[\overrightarrow{(\Gamma_i.B)}/\vec{X}])@y@\mathrm{id}_\Gamma\\
&\longleftrightarrow_T (C'[\overrightarrow{(\Gamma_i.B)}/\vec{X}])@\mathrm{id}_\Gamma\\
&= \widehat{C'}(\vec{B})
\end{aligned}$$

The prooftree then becomes the following

$$\dfrac{\dfrac{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid \Gamma' \vdash (y).C' : (y : D, \Gamma) \rightarrowtriangle *}{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid y : D, \Gamma' \vdash C' : \Gamma \rightarrowtriangle *}(*) \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{y : D, \Gamma', \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}\,) : \widehat{C'}(\vec{B})}\text{IdH.}$$

(*) This is the reverse of **(Param-Abstr)**.

- $\mathcal{D} =$

$$\dfrac{\begin{matrix}\mathcal{D}_1 \\ \sigma_k : \Delta_k \triangleright \Gamma\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n \rightarrowtriangle *, X : \Gamma \rightarrowtriangle * \mid \Delta_k \vdash D_k : *\end{matrix}}{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n \rightarrowtriangle * \mid \emptyset \vdash \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrowtriangle *}\textbf{(FP-Ty)}$$

From this we know $\Gamma' = \emptyset$

Calculated type actions:

$$\begin{aligned}
&\mu(Y : \widehat{\Gamma \rightarrowtriangle *}; \vec{\sigma}; \vec{D})(\vec{A}) \\
&= \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})[\overrightarrow{(\Gamma_i).A}/\vec{X}] @ \mathrm{id}_\Gamma \\
&= \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}]) @ \mathrm{id}_\Gamma \\
&\mu(Y : \widehat{\Gamma \rightarrowtriangle *}; \vec{\sigma}; \vec{D})(\vec{t}\,) \\
&= \mathrm{rec}^{\mu(Y:\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}])} \overrightarrow{(\Delta_k, x).\alpha_k @ \mathrm{id}_{\Delta_k} @ \widehat{D_k}(\vec{t}, x)} @ \mathrm{id}_\Gamma @ x \\
&\mu(Y : \widehat{\Gamma \rightarrowtriangle *}; \vec{\sigma}; \vec{D})(\vec{B}) \\
&= \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})[\overrightarrow{(\Gamma_i).B}/\vec{X}] @ \mathrm{id}_\Gamma \\
&= \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}]) @ \mathrm{id}_\Gamma
\end{aligned}$$

From the assumptions

$$\begin{aligned}
&X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid \emptyset \vdash \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrowtriangle * \\
&\Gamma_i, x : A_i \vdash t_i : B_i
\end{aligned}$$

We have to proof that in **Ctx**

$$\Gamma, x : \mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{B}]) @ \mathrm{id}_\Gamma$$

the expression

$$\mathrm{rec}^{\mu(Y:\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}])} \overrightarrow{(\Delta_k, y).\alpha_k @ \mathrm{id}_{\Delta_k} @ \widehat{D_k}(t, y)} @ \mathrm{id}_\Gamma @ x$$

has type

$$\mu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}]) @ \mathrm{id}_\Gamma$$

We can use the induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle *, Y : \Gamma_{n+1} \rightarrowtriangle * \mid \Delta_k \vdash D_k : * \qquad \Gamma_i, x : A_i \vdash t_i : B_i}{\Delta_k, x : \widehat{D_k}(\vec{A}, A_{n+1}) \vdash \widehat{D_k}(\vec{t}, y) : \widehat{D_k}(\vec{B}, B_{n+1})}$$

See A.1 for a proof of it.

- $\mathcal{D} =$

$$\frac{\overset{\mathcal{D}_1}{\sigma_k : \Delta_k \triangleright \Gamma} \qquad \overset{\mathcal{D}_2}{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n \rightarrowtriangle *, X : \Gamma \rightarrowtriangle * \mid \Delta_k \vdash D_k : *}}{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n \rightarrowtriangle * \mid \emptyset \vdash \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrowtriangle *} \ \textbf{(FP-Ty)}$$

From this we know $\Gamma' = \emptyset$.

Calculated type actions:

$$\begin{aligned}
&\widehat{\nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})}(\vec{A}) \\
&= \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})[\overrightarrow{(\Gamma_i).A}/\vec{X}]@\mathrm{id}_\Gamma \\
&= \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).A}/\vec{X}])@\mathrm{id}_\Gamma \\
&\widehat{\nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})}(\vec{t}) \\
&= \mathrm{corec}^{\nu(Y:\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}])} \overrightarrow{(\Delta_k, x)\widehat{D_k}(\vec{t}, x)[(\xi_k @\mathrm{id}_{\Delta_k} @x)/x]} @\mathrm{id}_\Gamma @x \\
&\widehat{\nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})}(\vec{B}) \\
&= \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D})[\overrightarrow{(\Gamma_i).B}/\vec{X}]@\mathrm{id}_\Gamma \\
&= \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}])@\mathrm{id}_\Gamma
\end{aligned}$$

From the assumptions

$$X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle * \mid \emptyset \vdash \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrowtriangle *$$
$$\Gamma_i, x : A_i \vdash t_i : B_i$$

We have to proof that in **Ctx**

$$\Gamma, x : \nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[(\Gamma_1).A/X])@\mathrm{id}_\Gamma$$

the expression

$$\mathrm{corec}^{\nu(Y:\Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}])} \overrightarrow{(\Delta_k, x)\widehat{D_k}(\vec{t}, x)[(\xi_k @\mathrm{id}_{\Delta_k} @x)/x]} @\mathrm{id}_\Gamma @x$$

has type

$$\nu(Y : \Gamma \rightarrowtriangle *; \vec{\sigma}; \vec{D}[\overrightarrow{(\Gamma_i).B}/\vec{X}])@\mathrm{id}_\Gamma$$

We can use the induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrowtriangle *, \dots, X_n : \Gamma_n \rightarrowtriangle *, Y : \Gamma_{n+1} \rightarrowtriangle * \mid \Delta_k \vdash D_k : * \qquad \Gamma_i, y_k : A_i \vdash t_i : B_i}{\Delta_k, y_k : \widehat{D_k}(\vec{A}, A_{n+1}) \vdash \widehat{D_k}(\vec{t}, y) : \widehat{D_k}(\vec{B}, B_{n+1})}$$

See A.1 for this proof.

□

A.1. Proofs for recursion and corecursion

$$\frac{\Gamma_1 \vdash \sigma : \Gamma_2 \qquad \Gamma_3 \vdash \tau : \Gamma_1}{\Gamma_3 \vdash \sigma \circ \tau : \Gamma_2}\ (*)$$

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta,\Gamma_k,y_k : A_k[C/X] \vdash g_k : C@\sigma_k\end{matrix}}{\Delta,\Gamma_k,x : A_k[\mu/X] \vdash g_k[\widehat{A_k}(\mathrm{rec}^{\mu}\overline{(\Gamma_k,y_k).g_k}@\mathrm{id}_\Gamma@x/y_k]}\ \mathrm{TyAct} \qquad \begin{matrix}\mathcal{D}\\ \Delta \vdash \tau : \Gamma_k\end{matrix} \qquad \begin{matrix}\mathcal{D}\\ \Delta \vdash u : A_k[\mu/X]\end{matrix}}{\Delta \vdash g_k[\widehat{A_k}(\mathrm{rec}^{\mu}\overline{(\Gamma_k,y_k).g_k}@\mathrm{id}_\Gamma@x)/y_k][\tau,u] : C@\sigma_k}$$

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ \Delta,\Gamma_k,y_k : \Delta_k[C/X] \vdash g_k : C@\sigma_k\end{matrix}}{\Delta \vdash \mathrm{rec}^{\mu}\overline{(\Gamma_k,y_k).g_k} : (\Gamma, x : \mu@\sigma_k) \rightarrowtriangle C@\sigma_k}\ \textbf{(Ind-E)} \qquad \dfrac{\begin{matrix}\mathcal{D}\\ \Gamma_k \vdash \sigma_k : \Gamma\end{matrix} \qquad \begin{matrix}\mathcal{D}\\ \Delta \vdash \tau : \Gamma_k\end{matrix}}{\Delta \vdash \sigma_k \circ \tau : \Gamma}\ (*)}{\Delta \vdash (rec^{\mu}\overline{(\Gamma_k,y_k).g_k}@(\sigma_k \circ \tau)) : (x : \mu@\sigma_k) \rightarrowtriangle C@\sigma_k}\ \textbf{(Inst)} \qquad \dfrac{\dfrac{\dfrac{}{\Delta \vdash \alpha_k^{\mu} : (\Gamma_k, y : A_k[\mu/X]) \rightarrowtriangle \mu@\sigma_k}\ \textbf{(Ind-I)} \qquad \begin{matrix}\mathcal{D}\\ \Delta \vdash \tau : \Gamma_k\end{matrix}}{\Delta \vdash \alpha_k^{\mu}@\tau : (y : A_k[\mu/X]) \rightarrowtriangle \mu@\sigma_k}\ \textbf{(Inst)} \qquad \begin{matrix}\mathcal{D}\\ \Delta \vdash u : A_k[\mu/X])\end{matrix}}{\Delta \vdash \alpha_k^{\mu}@\tau@u : \mu@\sigma_k}\ \textbf{(Inst)}}{\Delta \vdash (\mathrm{rec}^{\mu}\overline{(\Gamma_k,y_k).g_k}@(\sigma_k \circ \tau))@(\alpha_k^{\mu}@\tau@u) : C@\sigma_k}\ \textbf{(Inst)}$$

$$\dfrac{\dfrac{\overbrace{\vdash \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}^{C} : \Gamma \rightarrowtriangle * \qquad \dfrac{\dfrac{}{\Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma, \Delta_k, y_k : A_k[\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]/X] \vdash \alpha_k^{\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]} : (\Delta_k, y_k : A_k[\overline{(\Gamma_k).B_k}/\overline{X_k}]) \rightarrowtriangle \mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]@\sigma_k}\ \textbf{(Ind-I)} \qquad \Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma, \Delta_k, y_k : \overbrace{A_k[\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]/X]}^{=\widehat{A_k}(\overline{B},?)} \vdash \widehat{A_k}(\bar{t}, y_k) : \overbrace{A_k[\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]/X]}^{=\widehat{A_k}(\overline{B},?)}}{\Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma, \Delta_k, y_k : A_k[\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]/X] \vdash \alpha_k^{\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@\widehat{A_k}(\bar{t}, y_k) : \mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]@\sigma_k}\ \textbf{(Inst)}}{\Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma \vdash \mathrm{rec}^{\mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}\overline{(\Delta_k, y_k).\alpha_k^{\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@\widehat{A_k}(\bar{t}, y_k)} : (\Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma) \rightarrowtriangle \mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]@\mathrm{id}_\Gamma}\ \textbf{(Ind-E)} \qquad \dots}{\Gamma, x : \mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma \vdash \mathrm{rec}^{\mu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}\overline{(\Delta_k, y_k).\alpha_k^{\mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@\widehat{A_k}(\bar{t}, y_k)}@\mathrm{id}_\Gamma@x : \mu[\overline{(\Gamma_k).B_k}/\overline{X_k}]@\mathrm{id}_\Gamma}\ \textbf{(Inst)}$$

$$\dfrac{\dfrac{\overbrace{\vdash \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}^{C} : \Gamma \rightarrowtriangle * \qquad \Gamma, x : \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma, \Delta_k, y_k : \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\sigma_k \vdash \widehat{A_k}(\bar{t}, y_k)[(\xi_k^{\nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@y_k)/y_k] : A_k[\nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]/X]}{\Gamma, x : \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma \vdash \mathrm{corec}^{\nu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}\overline{(\Delta_k, y_k).\widehat{A_k}(\bar{t}, y_k)[(\xi_k^{\nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@y_k)/y_k]} : (\Gamma, x : \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma) \rightarrowtriangle \nu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}\ \textbf{(Ind-E)} \qquad \dots}{\Gamma, x : \nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]@\mathrm{id}_\Gamma \vdash \mathrm{corec}^{\nu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}\overline{(\Delta_k, y_k).\widehat{A_k}(\bar{t}, y_k)[(\xi_k^{\nu[\overline{(\Gamma_k).A_k}/\overline{X_k}]}@\mathrm{id}_{\Delta_k}@y_k)/y_k]}@\mathrm{id}_\Gamma@x : \nu[\overline{(\Gamma_k).B_k}/\overline{X_k}]}\ \textbf{(Inst)}$$

Bibliography

[Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. *arXiv preprint arXiv:1012.4896*, 2010.

[agd] Universe levels - agda 2.6.1.1 documentation.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. *ACM SIGPLAN Notices*, 48(1):27–38, 2013.

[BG16] Henning Basold and Herman Geuvers. Type theory based on dependent inductive and coinductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 327–336, 2016.

[BJSO19] David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decomposition diversity with symmetric data and codata. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[Chl13] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2013.

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

[Our08] Nicolas Oury, 06 2008. Message on the coq-clup maling list.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In *International Conference on Interactive Theorem Proving*, pages 499–514. Springer, 2014.