

1 **Implementation of Type Theory based on**
2 **dependent Inductive and Coinductive**
3 **Types**

4 Florian Engel

5 November 23, 2020

8



9 Masterarbeit

10 **Implementation of Type Theory based on**
11 **dependent inductive and coinductive types**

12 Eberhard Karls Universität Tübingen

13 Mathematisch-Naturwissenschaftliche Fakultät

14 Wilhelm-Schickard-Institut für Informatik

15 Programiersprachen

16 Florian Engel, florian.engel@student.uni-tuebingen.de, 2020

Bearbeitungszeitraum: von-bis

17

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

Zweitgutachter: Prof. Dr. Max Mustermann, Universität Tübingen

18 **Selbstständigkeitserklärung**

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

25 Florian Engel (Matrikelnummer 3860700), November 23, 2020

26 **Abstract**

27 Dependent types are a useful tool to restrict types even further than types of strongly
28 typed languages like Haskell. This gives us further type safety. With them, we can
29 also prove theorems. Coinductive types allow us to define types by their observations
30 rather than by their constructors. This is useful for infinite types like streams. In
31 many common dependently typed languages, like Coq and Agda, we can define
32 inductive types which depend on values and coinductive types but not coinductive
33 types, which depend on values.

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

Contents

| | | |
|----|--|-----------|
| 40 | 1. Introduction | 11 |
| 41 | 2. Coinductive Types | 15 |
| 42 | 3. Coinductive Types in dependent languages | 17 |
| 43 | 3.1. Coinductive Types in Coq | 19 |
| 44 | 3.1.1. Positive Coinductive Types | 19 |
| 45 | 3.1.2. Negative Coinductive Types | 21 |
| 46 | 3.2. Coinductive Types in Agda | 23 |
| 47 | 3.2.1. Positive Coinductive Types in Agda | 23 |
| 48 | 3.2.2. Negative Coinductive Types in Agda | 24 |
| 49 | 3.2.3. Termination Checking with Sized Types | 25 |
| 50 | 4. Type Theory based on dependent Inductive and Coinductive Types | 29 |
| 51 | 5. Implementation | 31 |
| 52 | 5.1. Abstract Syntax | 31 |
| 53 | 5.1.1. Declarations | 31 |
| 54 | 5.1.2. Expressions | 33 |
| 55 | 5.2. Substitution | 35 |
| 56 | 5.3. Typing rules | 36 |
| 57 | 5.3.1. Context rules | 38 |
| 58 | 5.3.2. Beta-equivalence | 38 |
| 59 | 5.3.3. Unit type and expression introduction | 39 |
| 60 | 5.3.4. Variable lookup | 39 |
| 61 | 5.3.5. Type and expression instantiation | 40 |
| 62 | 5.3.6. Parameter abstraction | 41 |
| 63 | 5.3.7. (co)inductive types | 41 |
| 64 | 5.3.8. Constructor and Destructor | 42 |
| 65 | 5.3.9. Recursion and Corecursion | 43 |
| 66 | 5.4. Evaluation | 44 |
| 67 | 6. Examples | 47 |
| 68 | 6.1. Terminal and Initial Object | 47 |
| 69 | 6.2. Natural Numbers and Extended Naturals | 48 |
| 70 | 6.3. Binary Product and Coproduct | 50 |

Contents

| | | |
|----|---|-----------|
| 71 | 6.4. Sigma and Pi Type | 51 |
| 72 | 6.5. Vectors and Streams | 52 |
| 73 | 7. Conclusion | 55 |
| 74 | A. Type action proof | 57 |
| 75 | A.1. Proofs for recursion and corecursion | 62 |

1. Introduction

In functional programming, we have functions that get input and produce 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 that define 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, therefore we write `_`. We are matching on the construction of the input values. Therefore, we call this method pattern matching. If we call this function somewhere in the code on values that 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 a lot of repetitive work. If Haskell would have coinductive types, this could be a lot less work. Coinductive types are types that are, 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 that 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 everywhere 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. Then, we add this constructor to every pattern matching on that type. For coinductive types it's 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 is structured as follows:

- Chapter 2 shows how coinductive types can be defined. Here, we will define the stream and function type, as well as 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 by [BG16]. With this theory we can then define coinductive types which depend on values. But we can not define types that 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.

144 • At last, we look at the examples from this paper [BG16] in the implemented
145 syntax. Here, we will see the reduction steps for recursion and corecursion.
146 We will conclude this section with the example of partial streams, which is a
147 coinductive type that depends on a value.

148 2. Coinductive Types

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

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

155 The **A** in the definition should be a concrete type. The type system in the paper
 156 doesn't have dependent types. What differentiates this from regular record types
 157 (for example in Haskell) is the recursive field **tail**. So they call it a recursive record.
 158 In a strict language without coinductive types we could never instantiate such a type
 159 because to do this we already need something of type **Stream A** to fill in the field **tail**.
 160 The paper defines copattern matching to remedy this. With the help of copattern
 161 matching, we can define functions that output expressions of type **Stream A**. As an
 162 example, we look at the definition of **repeat**. This function takes in a value of type
 163 **Nat** and generates a stream that just infinitely repeats it.

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

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

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

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

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

182 Here, we differentiate between our defined function $\mathbf{A} \rightarrow \mathbf{B}$ and \leadsto in the destructor.
 183 Constructor applications or, as is the case here, destructor applications are not the
 184 same as function applications, like in Haskell. In the paper $\mathbf{f} \mathbf{x}$ means **apply** $\mathbf{f} \mathbf{x}$. We
 185 will also use this convention in the following. In fact, we already used it in the defini-
 186 tions of the functions **repeat** and **nth**. $\mathbf{nth} \mathbf{0} \mathbf{x} = \mathbf{head} \mathbf{x}$ is just a nested copattern.
 187 We can also write it with **apply** like so: **apply** (**apply** **nth** $\mathbf{0}$) $\mathbf{x} = \mathbf{head} \mathbf{x}$. Here, we
 188 use currying. So the first **apply** is the sole observer of type $\mathbf{Stream} \mathbf{A} \rightarrow \mathbf{A}$ and the
 189 second of type $\mathbf{Nat} \rightarrow (\mathbf{Stream} \mathbf{A} \rightarrow \mathbf{A})$.

3. Coinductive Types in dependent languages

In this section, we will look at how coinductive types are implemented in dependently typed languages. In dependently typed languages types can depend on values. The classical example of such a type is the type for vectors. Vectors are like lists, 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 as 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 an empty vector called `Nil` and one to append an 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 as 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 in comparison to lists is that we can define a total function (a function which is defined for every input) that takes the head of a vector. This function can't be total for lists, because we cannot know if the input list is empty. An empty list has no head. For vectors, we can enforce this in Coq like follows:

```
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 pattern 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 follows:

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

212 That types can depend on terms makes it necessary to ensure that function
 213 terminate. Otherwise, type checking wouldn't be decidable. If we have a function
 214 $f : \text{Nat} \rightarrow \text{Nat}$ and we want to check a value a against a type $\text{Vec } (f \ 1)$ we have to
 215 know what $f \ 1$ evaluates to. So f has to terminate. We check termination in Coq
 216 via a structural decreasing argument. An argument is structural decreasing if it is
 217 structural smaller in a recursive call. Structural smaller means it is a recursive oc-
 218 currence in a constructor. As an example, we look at the definition of the natural
 219 numbers and the function for addition on them. We define the natural numbers in
 220 Coq like follows:

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.
```

221 0 is the constructor for 0 and S is the successor of its argument. Here, the recursive
 222 argument to S is structurally smaller than S applied to it i.e. n is structurally smaller
 223 than $S \ n$. Then, we can define addition like follows:

```
Fixpoint add (n m : nat) : nat :=
match n with
| 0 => m
| S p => S (add p m)
end.
```

224 In the recursive call, the first argument is structural decreasing. The expression
 225 p is smaller than the expression $S \ p$. So Coq accepts this definition. The classical
 226 example of a function where an argument is decreasing but not structural decreasing
 227 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.
```

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

236 For coinductive types termination means that functions that produce them should
 237 be productive. If a function is productive it produces in each step a new part of the
 238 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 these 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 accept our code. It rejects it because there is no argument which is structural decreasing. **x** stays always the same. Functions defined with **CoFixpoint** on the other hand only check 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.

```

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

```

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

```

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

279 We illustrate the purpose of the other conditions on an example taken from [Ch13].
 280 First, we implement the function `tl` like so:

```

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

```

281 This is just one normal pattern match on `Stream`. If we didn't have the other condi-
 282 tion we could define the following:

```

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

```

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

```

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

```

287 We can now simplify this to just:

```

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

```

288 After that `bad` isn't any more an argument to a constructor. Here, we can also see
 289 easily that the expression `cofix bad : Stream nat := bad` loops forever. So we never
 290 get the value at position 0.

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

```

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

```

296 We can now define a value of u with the following **Cofixpoint** like so:

```
Cofixpoint u : U := In u.
```

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

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

299 The same trick will be used to define **eq** which states that x is definitional equal to
 300 **force** x .

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

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

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

306 But u is not definitional equal to **In** u . As mentioned above expressions with a coin-
 307 ductive type are always values to prevent infinite evaluation. So **In** u is a value and
 308 u is also a value. But values are only definitional equal if they are exactly the same.
 309 The next section will solve this problem through negative coinductive types.

310 3.1.2. Negative Coinductive Types

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

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

314 Now we can define **repeat** over the fields of **Stream**.

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

315 To define **repeat** we must define what is the head of the constructed stream and its
 316 tail. The guard condition says now that corecursive occurrences must be guarded
 317 by a record field. We can see that the corecursive call **repeat** is a direct argument
 318 to the field **tl** of the corecursive type **Stream** A . This means Coq accepts the above
 319 definition. If we want to access parts of a stream we use the destructors **hd** and **tl**.
 320 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.

```

321 With negative coinductive types, we can't form the above-mentioned counterexample
 322 to subject reduction anymore, because we can't pattern match on negative types.
 323 Oury's example becomes.

```

CoInductive U := { out : U }.

```

324 U is now defined over its destructor **out**, instead of its constructor **in**. Then, **in**
 325 becomes just a function. In fact, it's just a definition because we don't recurse or
 326 corecure on it.

```

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

```

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

```

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

```

329 With coinductive types, it is now possible to define the pi type (the dependent
 330 function type).

```

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

```

331 The pi type is defined over its destructor **Apply**. If we evaluate **Apply** on a value
 332 of **Pi** (which is a function) and an argument, we get the result i.e. we apply the
 333 value to the function. It looks like the pi type becomes definable in Coq. But we are
 334 cheating. The type of **Apply** is already a pi type because we identify constructors
 335 and destructors with functions. We will see that the theory of the paper avoids this
 336 identification. To define a function we use **CoFixpoint**. As a simple nonrecursive,
 337 nondependent example we use the function **plus2**.

```

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

```

338 If we apply (i.e. call the destructor **Apply**) a **x** to **plus2** it gives back **S (S x)**. Which
 339 is twice the successor on **x**. So we add 2 to **x**. We use **_** here because **plus2** is not
 340 a dependent function i.e. the result type **nat** doesn't depend on the input value. To
 341 define functions with more than one argument we just use currying i.e. we use the
 342 type **Pi** as the second argument to **Pi**. For example, a 2-ary non-dependent function
 343 from **A** and **B** to **C** would have type **Pi A (fun _ => Pi B (fun _ => C))**. It would be
 344 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
    | 0 => { | Apply (m : nat) := m | }
    | S n' => { | Apply m := S (Apply _ _ (Apply _ _ plus n') m) | }
    end
  | }.

```

345 But Coq doesn't accept this definition. The guard condition is violated. The ex-
 346 pression `plus n'` is not a direct argument of the field `Apply`. The definition should
 347 terminate because we are decreasing `n` and the case for `0` is accepted. In the case of
 348 `0`, there is no recursive call.

349 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) | } | }.
```

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

353 3.2. Coinductive Types in Agda

354 In Agda coinductive types were first also introduced as positive types. In Section
 355 3.2.1 we will look at them in detail. In Section 3.2.2 we describe the correct way to
 356 implement coinductive types in Agda. There are functions which terminate but are
 357 rejected by the type checker. To allow more functions we can use a unique feature
 358 of Agda, sized types. They are described in Section 3.2.3.

359 3.2.1. Positive Coinductive Types in Agda

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

```
data Stream (A : Set) : Set where
  cons : A  $\rightarrow$   $\infty$  (Stream A)  $\rightarrow$  Stream A
```

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

```
#_ :  $\forall$  {a} {A : Set} a  $\rightarrow$  A  $\rightarrow$   $\infty$  A
b_ :  $\forall$  {a} {A : Set} a  $\rightarrow$   $\infty$  A  $\rightarrow$  A
```

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

```
repeat : {A : Set}  $\rightarrow$  A  $\rightarrow$  Stream A
repeat x = cons x (# (repeat x))
```

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

```

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

```

377 Here, we have to use **b** on the right-hand side of the second case, to force the compu-
 378 tation of the tail of the input stream. We have to do that because **nth** wants a stream.
 379 It doesn't want a suspended stream. Productivity on coinductive types like stream
 380 is checked by only allowing non decreasing recursive calls behind the **#** constructor.

381 3.2.2. Negative Coinductive Types in Agda

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

```

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

```

386 A Stream has 2 fields. The field **hd** is the head of the stream. It has type **A**. The
 387 field **tl** is the tail of the stream. It is another stream, so it has type **Stream A**. **tl** is
 388 a recursive field. So Agda wouldn't accept the definition without **coinductive**. A
 389 stream can never be empty. Every stream has a head (a field **hd**) and an empty
 390 stream wouldn't have a head. So the tail of a stream can never be empty. Therefore,
 391 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

```

392 We have to copattern match on every field of **Stream**, namely **hd** and **tl**. Because
 393 Agda is total it won't accept non-exhaustive (co)pattern matches like Haskell. First,
 394 we define what the head of **repeat x** is. We just repeat **x** infinitely often. So every
 395 element of the stream is **x**, including the head. Therefore, we just write **x**. In the
 396 second and last copattern we define what the tail of the stream is. The tail is just
 397 **repeat x**. Infinitely often repeated **x** is the same as **x** and then infinitely repeated **x**.
 398 We can use normal pattern matching and the destructors for functions that consume
 399 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. Agda doesn't define the first-class pi type like that. 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 τ 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. 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 from any function in the list for at least one input. As an example of such a function, 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 $x - y$ is smaller than $x + 1$, 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 were 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 these 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 the size i . The constructor **zero** is of arbitrary size i . The constructor **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, 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. Though the type is smaller than 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 its first input (here `x`), we also know that `(x - y)` has the same size as `x`. Therefore, `(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 a stream, the output stream's 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 stream's 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 streams.

```

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 content 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))

```

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

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

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

477 With sized types, we can define many total algorithm, which don't have a structurally
 478 decreasing argument, in a total language. In contrary to the Bove Capretta method
 479 [BC05], we don't have to change the structure of the algorithm.

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 $\Gamma \rightarrow *$. Here, Γ is a context which states to what terms we can apply the type. For example, we can apply A of kind $(x : B) \rightarrow *$ only to a term of type B . If we apply it to t of type B , we get a type of kind $*$. We write \rightarrow 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 $(x : A) \rightarrow B$ and apply it to a term s of type A we get a term of type B . We can also define our own types. $\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})$ is an inductive type and $\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})$ is a coinductive type. X is a variable that stands for the recursive occurrence of the type. It has the same kind $\Gamma \rightarrow *$ 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:

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

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 $\text{succ}@N$ and returns an A . The second destructor is the tail. It becomes also a stream of length $\text{succ}@N$. 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 .

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

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

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

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

515 5. Implementation

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

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

526 5.1. Abstract Syntax

527 In the following, we will scratch out the abstract syntax. In contrast to [BG16] we
528 can't write anonymous inductive and coinductive types. We will give every inductive
529 and coinductive type a name. They will be defined via declarations. In these declara-
530 tions, we will give, their constructors/destructors. They will also be given names. In
531 [BG16] they are anonymous. We can then refer to the previously defined types. We
532 will describe declarations in Section 5.1.1. We will also be able to bind expressions
533 to names. In Section 5.1.2 we will define the syntax of expressions. This will mostly
534 be in one to one correspondence with the syntax of [BG16]. Note however, that we
535 use the names of the constructors instead of anonymous constructors together with
536 their type and number. Also, the order of the matches in **rec** and **corec** is irrelevant.
537 We use the names of the Con/Destructors to identify them. In the following Section
538 6, we will see how the examples from the paper look in our concrete syntax.

539 5.1.1. Declarations

540 The abstract syntax is given in Figure 5.1. With the keywords **data** and **codata**
541 we define inductive and coinductive types respectively. After that, we will write
542 the name. We can only use names that aren't used already. Behind that, we can
543 give a parameter context. This is a type context. These types are not polymorphic.
544 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 | $:= x, y, z, \dots$ | Expression variables |
| TV | $:= X, Y, Z, \dots$ | Type expression variables |
| PV | $:= A, B, C, \dots$ | Parameter variables |
| EC | $:= \diamond$ $ (EV : TV, EV : TV)^*$ | Expression Context |
| PC | $:= \langle \rangle$ $ \langle (PV : EC \rightarrow \text{Set})^* \rangle$ | Parameter Context |
| $Decl$ | $:= \text{data } N \text{ } PC : (EC \rightarrow)? \text{ Set where}$ $(N : (EC \rightarrow)? \text{TypeExpr} \rightarrow N \text{Expr})^*$ $ \text{codata } N \text{ } PC : (EC \rightarrow)? \text{ Set where}$ $(N : (EC \rightarrow)? N \text{Expr}^* \rightarrow \text{TypeExpr})^*$ $ n \text{ } PC \text{ } EC = \text{Expr}$ | Declarations |

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 doesn't have 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 have 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 $\rho(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow *$ as follows:

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

- 563 • The $Expr^*$ are the $\vec{\sigma}$
- 564 • The first EC is Γ
- 565 • The other EC stand for $\Gamma_1, \dots, \Gamma_m$

566 To parse the abstract syntax we use Megaparsec. The parser generates an abstract
 567 syntax tree, which is given for declarations in Listing 1. The field **ty** in **ExprDef** is
 568 used later in type checking. The parser just fills them in with **Nothing**. Data and co-
 569 data definitions are both saved in **TypeDef**. The Haskell type **OpenDuctive** contains
 570 all the information for inductive and coinductive types. It corresponds to μ and
 571 ν in the paper. We use an **OpenDuctive** where the field **inOrCoin** is **IsIn** for μ
 572 and an **OpenDuctive** where the field **inOrCoin** is **IsCoin** for ν . The Haskell type
 573 **StrDef** ensures that the sigmas, as and gammals have the same length. We omit
 574 the implementation details for the parser because we are mainly focused on type
 575 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
                     , gammal :: Ctx
                     , strName :: Text
                     }

```

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

576 5.1.2. Expressions

577 The abstract syntax for expression is given in Figure 5.2. We will separate expres-
 578 sions in expressions for terms and expressions for types. There are given as regular
 579 expressions in **Expr** and **TypeExpr** respectively.

580 An **Expr** is either a **rec**, a **corec**, a con/destructor, an application @, the only
 581 primitive unit expression \diamond or a variable. With the keyword **rec** we can destruct
 582 an inductive type. We write **NParInst? to TypeExprp**, where **N** is a previously de-
 583 fined inductive type and **ParInst?** the instantiation of its parameter context, after
 584 **rec** to facilitate type checking. It says we want to destruct an inductive type to

| | | |
|------------|---|---|
| $ParInst$ | $:= \langle TypeExpr, TypeExpr \rangle^*$ | Instantiations for parameter contexts |
| $ExprInst$ | $:= (Expr, Expr)^*$ | Instantiations for expression contexts |
| $Expr$ | $:= \text{rec } N \text{ } ParInst? \text{ to } TypeExpr \text{ where}$ $\quad Match^*$ $ \text{ corec } TypeExpr \text{ to } N \text{ } ParInst? \text{ where}$ $\quad Match^*$ $ Expr @ Expr$ $ \Diamond$ $ EV$ $ n \text{ } ParInst \text{ } ExprInst$ | expression |
| $Match$ | $:= N \text{ } EV^* = Expr$ | match |
| $TypeExpr$ | $:= (EV : TypeExpr).TypeExpr$ $ TypeExpr @ Expr$ $ Unit$ $ TV$ $ N \text{ } ParInst?$ | Type expressions |

Figure 5.2.: Syntax for expressions

some other 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 **NParInst?** and the **TypeExpr** and list the destructors. All con/destructors have to be instantiated with all variables in the parameter contexts of their types. This is done by giving types of the expected kinds separated by ',' enclosed in \langle and \rangle . The variables are separated into local variables and global variables. Global variables refer to previously defined expressions. We have to fully instantiate their 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 that 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 with 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

```

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

we encounter a binder. This shifting is done in the module **ShiftFreeVars.hs**. We 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 substitutions, 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, there 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.

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

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

652 We also rewrite the rules which are already syntax-directed to rules which work on
 653 our syntax. We will mark semantic differences in the rewritten rules gray. We use
 654 variables $\Phi, \Phi', \Phi_1, \Phi_2, \dots$ for parameter contexts, $\Theta, \Theta', \Theta_1, \Theta_2, \dots$ for type variable
 655 contexts and $\Gamma, \Gamma', \Gamma_1, \Gamma_2, \dots$ for term variable contexts. The judgments in our rules
 656 are of one of the following form.

- 657 • $\Phi | \Theta | \Gamma \vdash \Theta'$ - The type variable context Θ' is well-formed in the combined
 658 context $\Phi | \Theta | \Gamma$.
- 659 • $\Phi | \Theta | \Gamma \vdash \Gamma'$ - The term variable context Γ' is well-formed in the combined
 660 context $\Phi | \Theta | \Gamma$.
- 661 • $\Phi | \Theta | \Gamma \vdash \Phi'$ - The parameter variable context Φ' is well-formed in the combined
 662 context $\Phi | \Theta | \Gamma$.
- 663 • $A \longrightarrow_T^* B$ - The type A fully evaluates to type B .
- 664 • $A \equiv_\beta B$ - The type A is computational equivalent to type B .
- 665 • $\Phi | \Theta | \Gamma \vdash A : \Gamma_2 \rightarrow *$ - The type A is well-formed in the combined context
 666 $\Phi | \Theta | \Gamma$ and can be instantiated with arguments according to context Γ_2 .
- 667 • $\Phi | \Theta | \Gamma \vdash t : \Gamma_2 \rightarrow A$ - The term t is well-formed in the combined context $\Phi | \Theta | \Gamma$
 668 and can be instantiated with arguments according to context Γ_2 . After this
 669 instantiation, it is of type A , where the arguments are substituted in A .
- 670 • $\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2$ - The context morphism σ is a well-formed substitution for Γ_2
 671 with terms in context Γ_1 in parameter context Φ .

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

674 In the module **TypeChecker** we will implement the following rules. It defines a monad
 675 **TI** which can throw errors and has a reader on the contexts in which we are type
 676 checking. To add something to a context we use the function **local**. This function
 677 gets a function to change the current content of the reader monad and executes a
 678 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.

$$\frac{}{\vdash \emptyset \text{ TyCtx}} \quad \frac{\vdash \Theta \text{ TyCtx} \quad \vdash \Gamma \text{ Ctx}}{\vdash \Theta, X : \Gamma \rightarrow * \text{ TyCtx}}$$

$$\frac{}{\vdash \emptyset \text{ Ctx}} \quad \frac{|\emptyset| \Gamma \vdash A : *}{\vdash \Gamma, x : A \text{ Ctx}}$$

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.

$$\frac{}{\vdash \emptyset \text{ ParCtx}} \quad \frac{\vdash \Phi \text{ ParCtx} \quad \vdash \Gamma \text{ Ctx}}{\vdash \Phi, X : \Gamma \rightarrow * \text{ ParCtx}}$$

These are structural the same rules like 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 $\Theta(X) \rightsquigarrow \Gamma \rightarrow *$ for looking up the type variable X in type context Θ yields type $\Gamma \rightarrow *$. We add 2 rules for looking up something in a type context. They are:

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

Here, Y and X are different variables.

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

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

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:

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

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 .

708 The rules are:

$$709 \quad \frac{\neg \exists B : A \longrightarrow_T B}{A \longrightarrow_T^* A} \quad \frac{A \longrightarrow_T B \quad B \longrightarrow_T^* C}{A \longrightarrow_T^* C}$$

710 \longrightarrow_T is defined in Section 5.4.

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

$$712 \quad \frac{A \longrightarrow_T^* A' \quad B \longrightarrow_T^* B' \quad A' \equiv_\alpha B'}{A \equiv_\beta B}$$

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

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

$$717 \quad \frac{}{\emptyset \equiv_\beta \emptyset} \quad \frac{\Gamma_1 \equiv_\beta \Gamma_2 \quad A \equiv_\beta B}{\Gamma_{1,x} : A \equiv_\beta \Gamma_{2,y} : B}$$

718 5.3.3. Unit type and expression introduction

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

$$720 \quad \frac{}{\vdash \top : *} (\top\text{-I}) \quad \frac{}{\vdash \diamond : \top} (\top\text{-I})$$

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

$$723 \quad \frac{}{\Phi | \Theta | \Gamma \vdash \text{Unit} : *} (\text{Unit-I}) \quad \frac{}{\Phi | \Theta | \Gamma \vdash \diamond : \text{Unit}} (\top\text{-I})$$

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

727 5.3.4. Variable lookup

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

731 The rule:

$$732 \quad \frac{\vdash \Theta \quad \text{TyCtx} \quad \vdash \Gamma \quad \text{Ctx}}{\Theta, X : \Gamma \rightarrow * \mid \emptyset \vdash X : \Gamma \rightarrow *} \text{TyVar-I}$$

733 gets rewritten to:

$$734 \quad \frac{\Theta(X) \rightsquigarrow \Gamma \rightarrow * \quad \vdash \Gamma_1 \quad \text{Ctx}}{\Phi | \Theta | \Gamma_1 \vdash X : \Gamma \rightarrow *} \text{TyVar-I}$$

735 The rule:

$$736 \frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A} \text{ (Proj)}$$

737 gets rewritten to:

$$738 \frac{\Gamma(x) \rightsquigarrow A}{\Phi \mid \Theta \mid \Gamma \vdash x : A} \text{ (Proj)}$$

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

$$740 \frac{\Phi(X) \rightsquigarrow \Gamma \rightarrow * \quad \vdash \Gamma_1 \quad \text{Ctx}}{\Phi \mid \Theta \mid \Gamma_1 \vdash X : \Gamma \rightarrow *} \text{ TyVar-I}$$

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

745 5.3.5. Type and expression instantiation

746 We can instantiate types and terms. The rule:

$$747 \frac{\Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \rightarrow * \quad \Gamma_1 \vdash t : B}{\Theta \mid \Gamma_1 \vdash A @ t : \Gamma_2[t/x] \rightarrow *} \text{ (Ty-Inst)}$$

748 for instantiating types gets rewritten to:

$$749 \frac{\Phi \mid \Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \rightarrow * \quad \Phi \mid \Theta \mid \Gamma_1 \vdash t : B' \quad B \equiv_\beta B'}{\Phi \mid \Theta \mid \Gamma_1 \vdash A @ t : \Gamma_2[t/x] \rightarrow *} \text{ (Ty-Inst)}$$

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

754 We also have a rule to instantiate terms. This rule:

$$755 \frac{\Gamma_1 \vdash t : (x : A, \Gamma_2) \rightarrow B \quad \Gamma_1 \vdash s : A}{\Gamma_1 \vdash t @ s : \Gamma_2[s/x] \rightarrow B[s/x]} \text{ (Inst)}$$

756 gets rewritten to:

$$757 \frac{\Phi \mid \Theta \mid \Gamma_1 \vdash t : (x : A, \Gamma_2) \rightarrow B \quad \Phi \mid \Theta \mid \Gamma_1 \vdash s : A' \quad A \equiv_\beta A'}{\Phi \mid \Theta \mid \Gamma_1 \vdash t @ s : \Gamma_2[s/x] \rightarrow B[s/x]} \text{ (Inst)}$$

758 These rules are similar to the rule for type instantiation. Here, we have to check (or
 759 infer) a term instead of a type. We also have to substitute s in the result type of t
 760 (in the case of types it's always $*$, which obviously has no free variables).

761 5.3.6. Parameter abstraction

762 The rule:

$$763 \frac{\Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \rightarrow *}{\Theta \mid \Gamma_1 \vdash (x).B : (x : A, \Gamma_2) \rightarrow *} \text{ (Param-Abstr)}$$

764 gets rewritten to:

$$765 \frac{\Phi \mid \Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \rightarrow *}{\Phi \mid \Theta \mid \Gamma_1 \vdash (x:\mathbf{A}).B : (x : A, \Gamma_2) \rightarrow *} \text{ (Param-Abstr)}$$

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

769 5.3.7. (co)inductive types

770 We have to separate the rule:

$$771 \frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Theta, X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : *}{\Theta \mid \emptyset \vdash \rho(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow *} \text{ (FP-Ty)}$$

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

$$774 \frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Phi \mid X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : * \quad \vdash \phi \text{ ParCtx}}{\vdash \text{data } X\langle\Phi\rangle : \Gamma \rightarrow \text{Set where; } \overline{\text{Constr}}_k : \Gamma_k \rightarrow A_k \rightarrow X\sigma_k} \text{ (FP-Ty)}$$

$$775 \frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Phi \mid X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : * \quad \vdash \phi \text{ ParCtx}}{\vdash \text{codata } X\langle\Phi\rangle : \Gamma \rightarrow \text{Set where; } \overline{\text{Destr}}_k : \Gamma_k \rightarrow X\sigma_k \rightarrow A_k} \text{ (FP-Ty)}$$

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

$$784 \frac{}{\Phi \vdash () : \Gamma_1 \triangleright \emptyset} \quad \frac{\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2 \quad \Phi \mid \Gamma_1 \vdash t : A[\sigma]}{\Phi \vdash (\sigma, t) : \Gamma_1 \triangleright (\Gamma_2, x : A)}$$

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

$$787 \frac{\Phi \mid \Theta \mid \Gamma' \vdash \vec{A} : \Gamma_i \rightarrow *}{\Phi \mid \Theta \mid \Gamma' \vdash X\langle\vec{A}\rangle : \Gamma[\vec{A}] \rightarrow *}$$

Here, X is a data or codata definition. The parser can decide if a variable is 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.

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

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:

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

gets rewritten to:

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

The rule for destructors:

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

gets rewritten to:

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

In the paper de/constructors are anonymous. They come together with their type. Therefore, 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:

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

gets rewritten to:

$$\frac{\vdash C : \Gamma \rightarrow * \quad \frac{\vdash \Gamma \equiv_\beta \Gamma'[\vec{D}]}{\vdash B_k \equiv_\beta (C@_{\sigma_k}[\vec{D}])} \quad \frac{\Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi'}{\Phi \parallel \Delta, \Gamma_k[\vec{D}], y_k : A_k[\vec{D}][C/X] \vdash g_k : B_k} \quad \Phi \mid \Theta \mid \Delta \vdash \text{rec } \mu\langle\vec{D}\rangle \text{ to } C; \text{Constr}_k \vec{x}_k y_k = g_k : (\Gamma, y : \mu[\vec{D}]@id_\Gamma) \rightarrow C@id_\Gamma}{\text{ (Ind-E)}}$$

We are recursing over some previously inductively defined type μ to some type C . These 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:

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

gets rewritten to:

$$\frac{\vdash C : \Gamma \rightarrow * \quad \frac{\vdash \Gamma \equiv_\beta \Gamma'[\vec{D}]}{\vdash B_k \equiv_\beta A_k[\vec{D}][C/X]} \quad \frac{\Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi'}{\Phi \parallel \Delta, \Gamma_k[\vec{D}], y_k : (C@_{\sigma_k}[\vec{D}]) \vdash g_k : B_k} \quad \Phi \mid \Theta \mid \Delta \vdash \text{corec } C \text{ to } \nu\langle\vec{D}\rangle; \text{Destr}_k \vec{x}_k y_k = g_k : (\Gamma, y : C@id_\Gamma) \rightarrow \nu[\vec{D}]@id_\Gamma}{\text{ (Coind-I)}}$$

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.

841 5.4. Evaluation

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

The first is a reduction on the type level (written \longrightarrow). It is defined as follows:

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

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

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

$$850 \frac{X : \Gamma_1 \rightarrow * \mid \Gamma'_2 \vdash C : \Gamma_2 \rightarrow * \quad \Gamma_1, x : A \vdash t : B}{\Gamma'_2, \Gamma_2, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

851 Here, we have a type C with a free type variable X and a term t of type B with a free
 852 term variable x of type A . If we use the action of this type on t we get a term with a
 853 type of this action on B . This term contains a free term variable x of type, the action
 854 applied to A . The type action is implemented in the module **TypeAction.hs**. Both
 855 the type action and the evaluation are done in the **Eval** monad. This monad has
 856 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:*

$$\begin{aligned} X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow * \\ \Gamma_i \vdash A_i : * \\ \Gamma_i \vdash B_i : * \\ \Gamma_i, x : A_i \vdash t_i : B_i \end{aligned}$$

Then, we define the type action on terms inductively over C .

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

And the type action on types as follows:

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

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

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

The reduction on recursion is defined as follows:

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

863 If we apply a recursion $\overrightarrow{\text{rec}(\Gamma_k, y_k).g_k}$ to this context morphism and a constructor
 864 $\alpha_k@ \tau @u$, which is fully applied, we lookup the case for this constructor. In this case,
 865 we substitute τ for the variables from Γ_k and u , where we apply the recursion to
 866 all recursive occurrences, for y_k . For this application, we need the type action. So a
 867 recursion is destructing an inductive type and all its recursive occurrences to another
 868 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 as follows:

$$\xi_k@ \tau @(\overrightarrow{\text{corec}(\Gamma_k, y_k).g_k@(\sigma_k \bullet \tau)@u}) > \widehat{A_k}(\overrightarrow{\text{corec}(\Gamma_k, y_k).g_k@id_\Gamma @x})[g_k/x][\tau, u]$$

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

873 6. Examples

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

877 6.1. Terminal and Initial Object

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

```
883 codata Terminal : Set where  
884 terminal = corec Unit to Terminal where @ ◇
```

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

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

```
896 data Initial : Set where  
897 exfalsum(C : Set) = rec Initial to C where
```

898 6.2. Natural Numbers and Extended Naturals

899 We use the approach of Peano to define natural numbers. Therefore, we use the
 900 inductive type **Nat** with the constructors **Zero** and **Suc**. **Zero** is just the number
 901 zero. Every constructor has to have an argument, which can contain a recursive
 902 occurrence. Every Type **A** is isomorphic to the function type **Terminal** \rightarrow **A**. So we
 903 use **Terminal** for this occurrence. **Suc** is the successor. So the meaning of **Suc n** is
 904 $n + 1$.

```
905 data Nat : Set where
906   Zero : Terminal  $\rightarrow$  Nat
907   Suc : Nat  $\rightarrow$  Nat
908   zero = Zero @  $\diamond$ 
909   one = Suc @ zero
```

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

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

915 We use it on one to see all cases.

```
916 id @ one = id @ (Succ @ zero)
917   > Succ @ n[ $\widehat{X}(\text{id @ x})/n$ ] [zero]
918   = Succ @  $\widehat{X}(\text{id @ x})$  [zero]
919   = Succ @ (id @ x)[zero]
920   = Succ @ (id @ zero)
921   = Succ @ (id @ (Zero @  $\diamond$ ))
922   > Succ @ (Zero @ u[ $\widehat{\text{Unit}}(\text{id @ x})/u$ ][ $\diamond$ ])
923   = Succ @ (Zero @ u[ $\widehat{\text{Unit}}(\text{id @ x})/u$ ][ $\diamond$ ])
924   = Succ @ (Zero @  $\widehat{\text{Unit}}(\text{id @ x})$ )[ $\diamond$ ]
925   = Succ @ (Zero @ x)[ $\diamond$ ]
926   = Succ @ (Zero @ x) = Succ @ zero = one
```

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

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

```
936 data Maybe(A : Set) : Set where
937   Nothing : Unit  $\rightarrow$  Maybe
938   Just : A  $\rightarrow$  Maybe
939   nothing(A) = Nothing(A) @  $\diamond$ 
940 codata Conat : Set where
941   Prec : Conat  $\rightarrow$  Maybe(Conat)
```


6.2. Natural Numbers and Extended Naturals

942 succ = corec Conat to Conat where
 943 Prec x = Just⟨Conat⟩ @ x

944 We now define the values zero and infinity.

945 zero = (corec Unit to Conat where
 946 {Prev x = nothing⟨Unit⟩}) @ ◇
 947 infinity = (corec Unit to Conat where
 948 {Prev x = Just⟨Conat⟩ @ x}) @ ◇

949 For **zero** the predecessor is absent, there is no predecessor of 0 in the natural num-
 950 bers, so we give pack **Nothing**. We then have to apply the **corec** to ◇ to get the value.
 951 The predecessor of **infinity** should also be **infinity**. We apply the **corec** to an-
 952 other **Conat**, so the **x** is also a **Conat**. We will now see that the predecessor on these
 953 values gives back the right value.

$$\begin{aligned}
 \text{Prev @ zero} &> \widehat{\text{Maybe}(X)} \left(\underbrace{\text{corec Unit to Conat where } \{ \text{Prev } x = \text{nothing}(\text{Unit}) \}}_{t_1} @ x \right) [\text{nothing}(\text{Unit})/x][\diamond] \\
 &= \text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
 &\quad \{ \text{Nothing } u = \text{Nothing}(\text{Conat}) @ \widehat{\text{Unit}}(t_1, u) \\
 &\quad \text{Just } c = \text{Just}(\text{Conat}) @ \widehat{X}(t_1, c) \} @ x [\text{nothing}(\text{Unit})/x][\diamond] \\
 &= \underbrace{\text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
 &\quad \{ \text{Nothing } u = \text{Nothing}(\text{Conat}) @ u \\
 &\quad \text{Just } c = \text{Just}(\text{Conat}) @ t_1 \}}_{t_2} @ \text{nothing}(\text{Unit}) \\
 &> \text{Nothing}(\text{Conat}) @ u [\widehat{\text{Unit}}(t_2 @ x)/u][\diamond] \\
 &= \text{Nothing}(\text{Conat}) @ u [x/u][\diamond] \\
 &= \text{Nothing}(\text{Conat}) @ \diamond
 \end{aligned}$$

$$\begin{aligned}
 \text{Prev @ infinity} &> \widehat{\text{Maybe}(X)} \left(\underbrace{\text{corec Unit to Conat where } \{ \text{Prev } x = \text{Just}(\text{Unit}) @ x \}}_{t_1} @ x \right) [\text{Just}(\text{Unit}) @ /x][\diamond] \\
 &= \text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
 &\quad \{ \text{Nothing } u = \text{Nothing}(\text{Conat}) @ \widehat{\text{Unit}}(t_1, u) \\
 &\quad \text{Just } c = \text{Just}(\text{Conat}) @ \widehat{X}(t_1, c) \} @ x [\text{Just}(\text{Unit}) @ /x][\diamond] \\
 &= \underbrace{\text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
 &\quad \{ \text{Nothing } u = \text{Nothing}(\text{Conat}) @ u \\
 &\quad \text{Just } c = \text{Just}(\text{Conat}) @ t_1 \}}_{t_2} @ \text{Just}(\text{Unit}) @ \\
 &> \text{Just}(\text{Conat}) @ t_1 [\widehat{\text{Unit}}(t_2 @ x)/x][\diamond] \\
 &= \text{Just}(\text{Conat}) @ t_1 [x/x][\diamond] \\
 &= \text{Just}(\text{Conat}) @ \text{infinity}
 \end{aligned}$$

954

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 product types. For example, if B depends on \mathbf{Nat} , we define the 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 \mathbf{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

$$\frac{(A : *, B : *) \parallel (x : A) \vdash \text{Snd} @ x : \text{Product}\langle A, B \rangle \text{ (a)} \quad (A : *, B : *) \parallel \vdash \text{Product}\langle A, B \rangle : * \quad (A : *, B : *) \parallel (y : B) \vdash \text{Fst} @ y : \text{Product}\langle A, B \rangle \text{ (b)}}{(A : *, B : *) \parallel \vdash \text{swap} : (p : \text{Product}\langle A, B \rangle) \rightarrow \text{Product}\langle B, A \rangle}$$

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

$$\frac{(A : *, B : *) \parallel (x : A) \vdash \text{Snd} : (x : A) \rightarrow \text{Product}\langle A, B \rangle \quad \frac{(x : A)(x) \rightsquigarrow A}{(x : A) \vdash x : A}}{(A : *, B : *) \parallel (x : A) \vdash \text{Snd} @ x : \text{Product}\langle A, B \rangle}$$

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 .

```

989 data Coproduct⟨A,B⟩ : Set where
990   Left  : A → Coproduct
991   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**.

```

996 data Sigma⟨A : Set , B : (x : A) → Set⟩ : Set where
997   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**.

```

1002 codata Pi⟨A : Set , B : (x : A) → Set⟩ : Set where
1003   Apply : (x : A) → Pi x → B

```

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

```

1006 id⟨A : Set⟩ = corec Unit to Pi⟨A,(v:A).A⟩ where
1007   { Apply v p = v } @ ◇

```

Evaluation on one goes as follows:

```

1009 apply = Apply⟨Nat,(v : Nat).Nat⟩
1010 one = S @ (Z @ )
1011 apply @ id⟨Nat⟩ @ one
1012 = apply @ one @ ((corec Unit to Pi⟨Nat,(x:Nat).Nat⟩ where
1013   Apply v p = v ) @ ◇)

```

$$\begin{aligned}
 1014 \quad & \succ \widehat{\text{Nat}} \left(\underbrace{\text{corec Unit to Pi where } \{ \text{Apply } v _ = v \} @ x}_t [v/x] [one, \diamond] \right)
 \end{aligned}$$

```

1015 = (rec Nat to Nat where
1016   Zero x = Zero @ (Unit(t,x))
1017   Succ x = Suc @ (Y(t,x)) @ x[v/x] [one, ◇]
1018 = (rec Nat to Nat where
1019   Zero x = Zero @ (Unit(t))
1020   Succ x = Suc @ x @ x[v/x] [one, ◇]
1021 = (rec Nat to Nat where
1022   Zero x = Zero @ (Unit())
1023   Succ x = Suc @ x @ x[v/x] [one, ◇]
1024 = (rec Nat to Nat where
1025   Zero x = Zero @ x
1026   Succ x = Suc @ x @ x[v/x] [one, ◇]
1027 = (rec Nat to Nat where
1028   Zero x = Zero @ x
1029   Succ x = Suc @ x @ v [one, ◇]

```

```

1030 = (rec Nat to Nat where
1031     Zero x = Zero @ x
1032     Succ x = Suc @ x) @ one
1033 = one

```

1034 6.5. Vectors and Streams

1035 Vectors are a standard example for dependent types. They are like lists, except their
 1036 type depends on their length. For example, a vector `[1;2]` has type `Vector<Nat> 2`,
 1037 because its length is 2. It has 2 constructors `Nil` and `Cons` like lists. `Nil` gives back
 1038 the empty vector. Because the length of the empty vector is zero its return type is
 1039 `Vector 0`. The second constructor `Cons` takes a natural number `k`, a value of type `A`
 1040 and a vector of length `k`, a `Vector k`. It returns a new vector. Its head is the first
 1041 argument and its tail the second. So the length of the result is one more than the
 1042 second argument. Therefore, it is `Vector (Suc k)`. In [BG16] the head and tail are
 1043 encoded in a pair.

```

1044 data Vector<A : Set> : (n:Nat) → Set where
1045   Nil : Unit → Vector zero
1046   Cons : (k:Nat, v:A) → Vector @ k → Vector (Suc @ k)
1047   nil<A : Set> = Nil<A : Set> @ 0

```

1048 The function `extend` takes a value `x` and extends it to a vector.

```

1049 extend<A : Set> =
1050   rec Vec<A> to ((x).Vec< A> @ (Suc x) where
1051     Nil u = Cons<A> @ x @ nil<A>
1052     Cons k v = Cons<A> @ (Suc @ k) @ v

```

1053 The type checking of this function goes as follows:

$$\begin{array}{c}
 (A : \text{Set}) \Vdash (x).(\text{Vec}\langle A \rangle @ (\text{Suc } @ x)) : (k : \text{Nat}) \rightarrow * \\
 (A : \text{Set}) \Vdash (u : A) \vdash \text{Cons}\langle A \rangle @ 0 @ (\text{Nil}\langle A \rangle @) : (x).(\text{Vec}\langle A \rangle @ (\text{Suc } @ x)) @ 0 \\
 \hline
 (k : \text{Nat}, v : (x).(\text{Vec } @ (\text{Suc } @ x)) @ k) \vdash \text{Cons}\langle A \rangle @ (\text{Suc } @ k) @ v : (x).(\text{Vec } @ (\text{Suc } @ x)) @ (\text{Suc } @ k) \\
 \vdash \text{extend}\langle A \rangle : (k : \text{Nat}, y : \text{Vec}\langle A \rangle @ k) \rightarrow (x).(\text{Vec}\langle A \rangle @ (\text{Suc } x)) @ k
 \end{array}$$

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

```

extend⟨Nat⟩ @1 @ (Cons⟨Nat⟩ @0 @0 @nil⟨Nat⟩)
= extend⟨Nat⟩ @ (Suc @k • 0) @ (Cons⟨Nat⟩ @0 @0 @nil⟨Nat⟩)
> Cons⟨Nat⟩ @ (Suc @k) @ v [X̂k(extend⟨Nat⟩ @n @x)/v] [0, nil⟨Nat⟩]
= Cons⟨Nat⟩ @ (Suc @k) @ v [X̂(extend⟨Nat⟩ @n @x)[k/n]/v] [0, nil⟨Nat⟩]
= Cons⟨Nat⟩ @ (Suc @k) @ v [extend @n @x[k/n]/v] [0, nil⟨Nat⟩]
= Cons⟨Nat⟩ @ (Suc @k) @ v [extend @k @x/v] [0, nil⟨Nat⟩]
= Cons⟨Nat⟩ @ (Suc @k) @ (extend @k @x) [0, nil⟨Nat⟩]
= Cons⟨Nat⟩ @ (Suc @0) @ (extend @0 @ (nil⟨Nat⟩))
= Cons⟨Nat⟩ @1 @ (extend @0 @ (Nil⟨Nat⟩ @))
> Cons⟨Nat⟩ @1 @ (Cons⟨Nat⟩ @0 @ (Nil⟨Nat⟩ @)) [Unit(extend @k @x)/u] [◇]
= Cons⟨Nat⟩ @1 @ (Cons⟨Nat⟩ @0 @ (Nil⟨Nat⟩ @x)) [◇]
= Cons⟨Nat⟩ @1 @ (Cons⟨Nat⟩ @0 @ (Nil⟨Nat⟩ @))

```

1055 Here, we write 1 for **Suc @ (Zero @)** and 0 for **Zero @ ◇**.

1056 With the help of extended naturals, we can define partial streams. Those are streams
 1057 that depend on their definition depth. Like non-dependent streams, they are coin-
 1058 ductive and have 2 destructors for head and tail.

```

1059 codata PStr⟨A : Set⟩: (n : ExNat) → Set where
1060   hd : (k : ExNat) → PStr⟨A⟩ (succE k) → A
1061   tl : (k : ExNat) → PStr⟨A⟩ (succE k) → PStr⟨A⟩ @ k

```

1062 These streams are like vectors except they also can be infinite long. This is in contrary
 1063 to non-dependent streams. A non-dependent stream could not be of length zero.
 1064 Because then a call of **hd** and **tl** on it wouldn't be defined. In the dependent case,
 1065 the type checker wouldn't allow such a call because **hd** and **tl** expect streams which
 1066 are at least of length one. We can then define **repeat**.

```

1067 repeat⟨A : Set⟩(x : A, n : Conat) =
1068   corec (n : Conat).Unit to PStr⟨A⟩ where
1069     { Hd k s = x
1070       ; Tl k s = ◇ } @ n @ ◇

```

1071 This function gets a value and an extended natural number. It generates a constant
 1072 partial stream of that value with the number as its length.

1073 7. Conclusion

1074 We have implemented a dependent type theory with inductive and coinductive types.
1075 In this theory, contrary to Coq and Agda, coinductive types can also depend on val-
1076 ues. Contrary to the theory of the paper we can define schemata like `Maybe⟨A : Set⟩`
1077 where `A` can be an arbitrary type of kind `Set`.

1078 One downside is that we don't have universes. This prevents type polymorphism.
1079 Further work needs to be done to solve this. Another problem is, that each con-
1080 structor or destructor has at least one argument. The argument with the recursive
1081 occurrence. For example, we have to apply a unit to the constructors of a boolean
1082 type. We could allow recursive occurrences in the contexts of the constructors and
1083 destructors. This makes it possible to remove the argument with the recursive oc-
1084 currence. Then we have to change the evaluation rules.

1085 Our system allowed us to define the (depended) function type. Therefore, we don't
1086 have it as a primitive expression. We are hopeful, that in the future we get a more
1087 mainstream language, like Coq or Agda, where the dependet function is definable.
1088 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

- $\Gamma = \epsilon$:

$$A \longleftrightarrow_T A$$

- $\Gamma = x : B, \Gamma'$:

$$(x : B, \Gamma').A@x@id_{\Gamma'} \longrightarrow_p (\Gamma').A@id_{\Gamma'}[x/x] = (\Gamma').A@id_{\Gamma'} \xleftrightarrow{IdH}_T A$$

□

Theorem 2. *The following rule holds*

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

Proof. We show this by induction on t

□

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

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

Proof. First we will generalize the rule to

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

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

- $\mathcal{D} = \frac{}{\vdash \top : *} \text{ (}\tau\text{-I)}$

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

$$1103 \quad \frac{\vdash \top : *}{x : \top \vdash x : \top} \text{ (Proj)}$$

$$1104 \quad \bullet \mathcal{D} = \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n-1} : \Gamma_{n-1}} \text{ TyCtx} \quad \frac{\mathcal{D}_2}{\Gamma_n \text{ Ctx}} \text{ TyVar-I}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \emptyset \vdash X_n : \Gamma_n \rightarrow *}$$

Again we calculate the type actions

$$\begin{aligned} \widehat{X_n}(\vec{A}) &= X_n[(\Gamma_i).\vec{A}/\vec{X}]@id_{\Gamma_n} = X_n[(\Gamma_n).A_n/X_n]@id_{\Gamma_n} = (\Gamma_n).A_n@id_{\Gamma_n} \\ \widehat{X_n}(\vec{t}) &= t_n \\ \widehat{X_n}(\vec{B}) &= X_n[(\Gamma_i).\vec{B}/\vec{X}]@id_{\Gamma_n} = X_n[(\Gamma_n).B_n/X_n]@id_{\Gamma_n} = (\Gamma_n).B_n@id_{\Gamma_n} \end{aligned}$$

1105 We know from the first premise that $\Gamma = \Gamma_n$ and $\Gamma' = \emptyset$

1106 Here, we got the proof tree

$$1107 \quad \frac{\frac{\Gamma_n, x : A \vdash t : B}{\Gamma_n, x : (\Gamma_n).A@id_{\Gamma_n} \vdash t : B} \text{Thrm. 1} \quad \frac{A \longleftrightarrow_T (\Gamma_n).A@id_{\Gamma_n}}{\Gamma_n, x : (\Gamma_n).A@id_{\Gamma_n} \vdash t_n : (\Gamma_n).B@id_{\Gamma_n}} \text{Thrm. 2}}{B \longleftrightarrow_T (\Gamma_n).B@id_{\Gamma_n}} \text{Conv}$$

$$1108 \quad \bullet \mathcal{D} = \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrow *} \quad \frac{\mathcal{D}_2}{\Gamma_n \text{ Ctx}}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n+1} : \Gamma_{n+1} \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} \text{ (TyVar-Weak)}$$

1109 Here, we got the proof tree

$$1110 \quad \frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n+1} : \Gamma_{n+1} \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} (*)}{\Gamma', \Gamma, x : \underbrace{\widehat{C}(\vec{A})}_{\equiv \widehat{C}(\vec{A}, A_{n+1})} \vdash \underbrace{\widehat{C}(\vec{t})}_{\equiv \widehat{C}(\vec{t}, t_{n+1})} : \underbrace{\widehat{C}(\vec{B})}_{\equiv \widehat{C}(\vec{B}, B_{n+1})}}_{\Gamma_i, x : A_i \vdash t_i : B_i} \text{ IdH.}$$

1111 (*) Here, we undo (TyVar-Weak)

1112 (**) X_{n+1} doesn't occur free in C, otherwise \mathcal{D}_1 wouldn't be possible

1113 (***) Case for (TyVar-Weak) of type actions on terms

1114 $\bullet \mathcal{D} =$

$$1115 \quad \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrow *} \quad \frac{\mathcal{D}_2}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma', y : D \vdash C : \Gamma \rightarrow *} \text{ (Ty-Weak)}$$

1116 Here, we got the proof tree

$$1117 \quad \frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma', y : D \vdash C : \Gamma \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} (*)}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})} \text{ IdH.} \quad \frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}{\Gamma', \Gamma, x : \widehat{C}(\vec{A})y \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})} \text{ (Term-Weak)}$$

1119 (*) Here, we undo **(Ty-Weak)**

$$1120 \bullet \mathcal{D} = \frac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrow * \quad \Gamma' \vdash s : D}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C' @s : \Gamma \rightarrow *} \text{ (Ty-Inst)}$$

1121 Then, we got the following induction hypothesis

$$1122 \frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrow * \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', y : D, \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})}$$

Calculated type actions:

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

$$\widehat{C' @s}(\vec{t}) = \widehat{C'}(\vec{t})[s/y]$$

$$\widehat{C' @s}(\vec{B}) = C' @s[(\Gamma_i). \vec{B} / \vec{X}] @id_\Gamma = C'[(\Gamma_i). \vec{B} / \vec{X}] @s @id_\Gamma = \widehat{C'}(\vec{B})[s/y]$$

1123 We then got the following proof tree

$$1124 \frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma'_2 \vdash C' @s : \Gamma_2[s/y] \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma'_2 \vdash C' : (y : D, \Gamma_2) \rightarrow *} (*) \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\frac{\Gamma'_2, y : D, \Gamma_2, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})}{\Gamma'_2, \Gamma_2[s/y], x : \widehat{C'}(\vec{A})[s/y] \vdash \widehat{C'}(\vec{t})[s/y] : \widehat{C'}(\vec{B})[s/y]}} \text{ IdH.}$$

1125 (*) This is the reverse of **(Ty-Inst)**.

$$1126 \bullet \mathcal{D} = \frac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma', y : D \vdash C' : \Gamma \rightarrow *}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash (y). C' : (y : D, \Gamma) \rightarrow *} \text{ (Param-Abstr)}$$

Calculated type actions:

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

1127 The proof tree then becomes the following

Appendix A. Type action proof

$$\begin{array}{c}
1128 \quad \frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash (y).C' : (y : D, \Gamma) \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid y : D, \Gamma' \vdash C' : \Gamma \rightarrow *} (*) \\
1129 \quad \frac{\Gamma_i, x : A_i \vdash t_i : B_i}{y : D, \Gamma', \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})} \text{IdH.}
\end{array}$$

1130 (*) This is the reverse of **(Param-Abstr)**.

1131 • $\mathcal{D} =$

$$\begin{array}{c}
1132 \quad \frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \sigma_k : \Delta_k \triangleright \Gamma \quad X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow *, X : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow * \mid \emptyset \vdash \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow *} \text{(FP-Ty)}
\end{array}$$

1133 From this we know $\Gamma' = \emptyset$

Calculated type actions:

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

From the assumptions

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

We have to proof that in **Ctx**

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

the expression

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

has type

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

1134 We can use the induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow *, Y : \Gamma_{n+1} \rightarrow * \mid \Delta_k \vdash D_k : * \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\Delta_k, x : \widehat{D}_k(\vec{A}, A_{n+1}) \vdash \widehat{D}_k(\vec{t}, y) : \widehat{D}_k(\vec{B}, B_{n+1})} \quad 1135$$

1136 See A.1 for a proof of it.

1137 • $\mathcal{D} =$

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \sigma_k : \Delta_k \triangleright \Gamma \quad X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow *, X : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow * \mid \emptyset \vdash \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow *} \text{ (FP-Ty)} \quad 1138$$

1139 From this we know $\Gamma' = \emptyset$.

Calculated type actions:

$$\begin{aligned} & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{A}) \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{A} / \vec{X}]@id_\Gamma \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{A} / \vec{X}]@id_\Gamma \\ & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{t}) \\ &= \text{corec}^{\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]}(\Delta_k, x) \widehat{D}_k(\vec{t}, x)[(\xi_k @ id_{\Delta_k} @ x) / x]@id_\Gamma @ x \\ & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{B}) \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma \end{aligned}$$

From the assumptions

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

We have to proof that in **Ctx**

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

the expression

$$\text{corec}^{\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]}(\Delta_k, x) \widehat{D}_k(\vec{t}, x)[(\xi_k @ id_{\Delta_k} @ x) / x]@id_\Gamma @ x$$

has type

$$\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma$$

1140 We can use the induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow *, Y : \Gamma_{n+1} \rightarrow * \mid \Delta_k \vdash D_k : * \quad \Gamma_i, y_k : A_i \vdash t_i : B_i}{\Delta_k, y_k : \widehat{D}_k(\vec{A}, A_{n+1}) \vdash \widehat{D}_k(\vec{t}, y) : \widehat{D}_k(\vec{B}, B_{n+1})} \quad 1141$$

1142 See A.1 for this proof.

1143

□

A.1. Proofs for recursion and corecursion

1144

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

1145

$$\frac{\mathcal{D} \quad \Delta, \Gamma_k, y_k : A_k[C/X] \vdash g_k : C@_{\sigma_k} \quad \text{TyAct} \quad \mathcal{D} \quad \Delta \vdash \tau : \Gamma_k \quad \Delta \vdash u : A_k[\mu/X]}{\Delta, \Gamma_k, x : A_k[\mu/X] \vdash g_k[\widehat{A_k}(\text{rec}^\mu(\Gamma_k, y_k) \cdot g_k @ \text{id}_\Gamma @ x) / y_k] \rightarrow \text{id}_\Gamma @ x / y_k} \quad \Delta \vdash g_k[\widehat{A_k}(\text{rec}^\mu(\Gamma_k, y_k) \cdot g_k @ \text{id}_\Gamma @ x) / y_k] : C@_{\sigma_k}$$

1146

$$\frac{\mathcal{D} \quad \Delta, \Gamma_k, y_k : \Delta_k[C/X] \vdash g_k : C@_{\sigma_k} \quad (\text{Ind-E}) \quad \frac{\mathcal{D} \quad \Gamma_k \vdash \sigma_k : \Gamma \quad \Delta \vdash \tau : \Gamma_k}{\Delta \vdash \sigma_k \circ \tau : \Gamma} (*)}{\Delta \vdash \text{rec}^\mu(\Gamma_k, y_k) \cdot g_k @ (\sigma_k \circ \tau) : C@_{\sigma_k}} \quad \Delta \vdash (\text{rec}^\mu(\Gamma_k, y_k) \cdot g_k @ (\sigma_k \circ \tau)) @ (\alpha_k^\mu @ \tau @ u) : C@_{\sigma_k} \quad \Delta \vdash \alpha_k^\mu @ \tau @ u : \mu @_{\sigma_k} \quad \mathcal{D} \quad \Delta \vdash \tau : \Gamma_k \quad (\text{Inst}) \quad \Delta \vdash u : A_k[\mu/X] \quad (\text{Inst})$$

1147

$$\frac{\text{C} \quad \frac{\Gamma, x : \mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k} @ \text{id}_\Gamma, \Delta_k, y_k : A_k[\mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}] \vdash q_k^{\mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}} : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k} @ \alpha_k \quad \Gamma, x : \mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k} @ \text{id}_\Gamma, \Delta_k, y_k : A_k[\mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}] \vdash q_k^{\mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}} @ \text{id}_{\Delta_k} @ \widehat{A_k}(\vec{t}, y_k) : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k} @ \alpha_k}{\Gamma, x : \mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k} @ \text{id}_\Gamma \vdash \text{rec}^{\mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}} @ \text{id}_{\Delta_k} @ \widehat{A_k}(\vec{t}, y_k) : \mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k} @ \text{id}_\Gamma} \quad \Gamma, x : \mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k} @ \text{id}_\Gamma \vdash \text{rec}^{\mu(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}} @ \text{id}_{\Delta_k} @ \widehat{A_k}(\vec{t}, y_k) @ \text{id}_\Gamma @ \alpha_k : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k} @ \text{id}_\Gamma \quad \dots \quad (\text{Inst})$$

1148

$$\frac{\text{C} \quad \frac{\Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] : \Gamma \rightarrow * \quad \Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \text{id}_\Gamma, \Delta_k, y_k : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \sigma_k \vdash \widehat{A_k}(\vec{t}, y_k) [(\xi_k^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k) / y_k] : A_k[v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] / X]}{\Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \text{id}_\Gamma \vdash \text{corec}^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k / y_k : (\Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \text{id}_\Gamma @ \alpha_k : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}) \rightarrow v[(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k}]} \quad \Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \text{id}_\Gamma \vdash \text{corec}^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k / y_k [(\xi_k^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k) / y_k] @ \text{id}_\Gamma @ \alpha_k : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k} \quad \dots \quad (\text{Inst})$$

1149

$$\frac{\Gamma, x : v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}] @ \text{id}_\Gamma \vdash \text{corec}^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k / y_k [(\xi_k^{v[(\Gamma_k) \cdot \widehat{A_k} / \widehat{X_k}]} @ \text{id}_{\Delta_k} @ y_k) / y_k] @ \text{id}_\Gamma @ \alpha_k : \mu(\Gamma_k) \cdot \widehat{B_k} / \widehat{X_k} \quad \dots \quad (\text{Inst})$$

1150

1151 Bibliography

- 1152 [Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. *arXiv*
1153 *preprint arXiv:1012.4896*, 2010.
- 1154 [agd] Universe levels - agda 2.6.1.1 documentation.
- 1155 [APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Co-
1156 patterns: programming infinite structures by observations. *ACM SIG-*
1157 *PLAN Notices*, 48(1):27–38, 2013.
- 1158 [BC05] Ana Bove and Venanzio Capretta. Modelling general recursion in type
1159 theory. *Mathematical Structures in Computer Science*, 15(4):671–708,
1160 2005.
- 1161 [BG16] Henning Basold and Herman Geuvers. Type theory based on depen-
1162 dent inductive and coinductive types. In *Proceedings of the 31st Annual*
1163 *ACM/IEEE Symposium on Logic in Computer Science*, pages 327–336,
1164 2016.
- 1165 [BJSO19] David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decom-
1166 position diversity with symmetric data and codata. *Proceedings of the*
1167 *ACM on Programming Languages*, 4(POPL):1–28, 2019.
- 1168 [Chl13] Adam Chlipala. *Certified programming with dependent types: a pragmatic*
1169 *introduction to the Coq proof assistant*. MIT Press, 2013.
- 1170 [DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless
1171 dummies, a tool for automatic formula manipulation, with application to
1172 the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*,
1173 volume 75, pages 381–392. North-Holland, 1972.
- 1174 [Our08] Nicolas Oury, 06 2008. Message on the coq-clup mailing list.
- 1175 [ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq.
1176 In *International Conference on Interactive Theorem Proving*, pages 499–
1177 514. Springer, 2014.